Generalized Additive Models for Large datasets: spatial-temporal modelling of the UK's Daily Black Smoke (1961 - 2005)

Li, Zheyuan

Award date:
2019

Awarding institution:
University of Bath

Generalized Additive Models for Large datasets: spatial-temporal modelling of the UK's Daily Black Smoke (1961 - 2005)

submitted by

Zheyuan Li

for the degree of Doctor of Philosophy

of the

University of Bath

Department of Mathematical Sciences

April, 2018

Abstract

The UK Black Smoke monitoring network has produced daily particulate air pollution data from a network of up to 1200 monitoring stations over several decades, resulting in 10 million measurements in total. Spatial-temporal modelling of the data is desirable for accurate trend / seasonality estimation and mapping and to provide daily exposure estimates for epidemiological cohort studies. Generalized additive models offer one way to do this if we can deal with the data volume and model size. This thesis will develop computation method for estimating generalized additive models having $O(10^4)$ coefficients and $O(10^8)$ observations. The strategy combines 3 elements: (i) fine scale discretization of covariates, (ii) an efficient approach to restricted likelihood optimization, that avoids computation of numerically awkward log determinant terms and (iii) restricted likelihood optimization algorithms that make good use of numerical linear algebra methods with high performance and good parallel scaling on mordern multi-core machines. The new method enables us to estimate spatial-temporal models for daily Black Smoke data over the last four decades at a daily resolution which had once been infeasible. A spatial-temporal dataset of daily Black Smoke is also produced on a grid of 5km $\times$ 5km resolution. Our prediction is shown to suffer from little extrapolation and bias.

Contents

Introduction

Since the *Industrial Revolution*, problems associated with air pollution worsened in many countries. During the first half of the 20th century, major pollution episodes occurred in London, notably in 1952 an episode of fog, in which levels of Black Smoke (arising from the incomplete combustion of carbonaceous matter: solid fuels, fuel oils and waste materials, and is especially hazardous to human health in combination with other pollutants which adhere to the particulate surfaces) exceeded 4500 μgm^{-3}, was associated with 4000 excess deaths (Ministry of Health, 1954). Other early episodes, which were caused by a combination of industrial pollution sources and adverse weather conditions, and resulted in large numbers of deaths among the surrounding populations include those in the Meuse valley (Firket, 1936) and the US (Ciocco and Thompson, 1961). Attempts to measure levels of air pollution in a regular and systematic way arose as a result of these episodes.

Following the first *Clean Air Act* in 1952 and a few years to standardize measurement methods (British Standard 1747), the UK established in 1961 the world's first co-ordinated national air pollution monitoring network, the *National Survey of Air Pollution* (Clifton, 1964), to monitor Black Smoke and SO_2, with objectives to **1)** monitor progress of Clean Air Acts, and provide the technical basis for future legislation; **2)** monitor progress of Clean Air Acts, and provide the technical basis for future legislation; **3)** provide a consistent body of data for research (For example, effects of climate, topography, industrialization, population density, fuel utilization and urban development can be assessed on pollution levels. Effects of pollution levels on health can also be examined). Since then, the network had worked for 45 years, with a peak size of over 1200 sites per year, producing over 10 million daily measurements.

Because of the size of the dataset, previous statistical modelling with Black Smoke has focussed on modelling time or space averages pollution levels. Fanshawe et al. (2008) studied weekly Black Smoke in Newcastle area (with 20 sites) between 1962 and 1991; Dadvand et al. (2011) studied weekly Black Smoke in North East (a governmental region in England) between 1985 and 1996; Gulliver et al. (2011) produced spatial mapping over Great Britain on a yearly basis between 1962 and 1991; Shaddick and Zidek (2014) studied yearly averaged Black Smoke over 1466 sites across Great Britain between 1966 and 1996. Aggregated air pollution data are not entirely satisfactory from an epidemiological perspective. The variations in pollution are so rapid that even a week is too long a period for true contrasts and comparisons to be appreciated. Besides, acute respiratory disease is usually sensitive to exposure to high levels of pollution over short time periods, and such exposure can be completely hidden in an annual average. Retrospective cohort studies, for example, really require estimates of exposure at the daily level.

There could be several ways to obtain a daily estimate. The simplest one is the running mean, but this only gives estimates for a specific station or location, with no information elsewhere. For spatial prediction purpose, then a spatial smoothing on a daily basis may serve well, but this obscures the variation in time. Particularly, the network itself was changing all the time. There had been nearly 3000 historical stations, but the network only retained a good density over the island for 15 years (1966 - 1980) with about 1000 sites per year. To mitigate these effects, a full spatial-temporal model is most likely to yield reasonable predictions.

At the start of my PhD programme, it was not clear what a model would be like for Black Smoke and what complexity it might have. Preliminary modelling started from modelling small subsets of the dataset, like time series analysis, annual mean, etc. It soon turned out that many effects, like trend, seasonality and temperature effects all vary with space. Building models for these interactions results in great number of parameters. Given such complexity, model representation and data storage are readily huge challenges. Conventional estimation procedure requires "first generating and storing everything" is clearly infeasible. Some estimation routines, like the `bigglm` (Miller, 1992) or `speedglm` (Enea, 2009), even if they do produce smaller fitted model object or a faster estimation, still require storing the whole design matrix for all data and all parameters. In this aspect, the `bam` function from `R` package `mgcv`, designed for estimating generalized additive models (GAMs) with large datasets, gives a better strategy. Wood et al. (2015) proposed a way to split the huge design matrix into smaller, storable chunks and work with them one at a time. At first glance, this shares the same idea with `bigglm`, but a key feature of `bam` is that these chunks are generated "on the fly". Once a chunk is processed, it is discarded, making room for the next chunk. The memory footprint is bounded by the size of a chunk, independent of the size of the whole model matrix. For this reason, building GAMs was chosen as the primary method for model development.

This thesis will be structured as follows.

Chapter 1 and Chapter 2 are gentle introductions to smoothing problems and additive models via splines. The former focuses on explaining basic concepts, and the latter demonstrates how additive models can be used in practice via simulated toy datasets.

Chapter 3 conducts some preliminary modelling of Black Smoke by investigating the dataset from different angles: time series, annual mean and daily data from a single year. It turned out that black smoke models can end up with 10000 or even more regression coefficients, making a big computational hurdle for practical model development with GAM. While some statistical questions were also observed during model development, they all gave way to improving GAM fitting methods as without being able to fit a model as fast as possible, it is impractical to do for example, cross-validation.

To see where GAM fitting methods can be improved, it is a prerequisite to understand in depth how exisiting GAM computational engine works. Chapter 4 is a thorough walkthrough on this.

Chapter 5 starts the first round of performance upgrade of GAM fitting. Many steps are taken to progressively speedup GAM fitting, by the end of which, the computational hurdle previously seen in Chapter 3 are all overcome.

Chapter 6 revisits the preliminary modelling of Black Smoke. Some unanswered questions in Chapter 3 will be addressed.

Chapter 7 makes an attempt at building a daily model for the complete Black Smoke dataset. However, even after using 22000+ parameters, the model is still very inadequate. I eventually give up building models and focus on developing new GAM computation methods for large datasets.

Chapter 8 is a second round of performance upgrading of GAM fitting methods. A novel covariate discretization procedure, packed storage for model matrix, and a set of discrete algorithms to perform required matrix cross-product $\boldsymbol{X'X}$ are developed. A discrete algorithm is actually not as abstract as it sounds. For example, suppose we have a vector of repeated values, like 2, 4, 4, 5, 2, 5, 2 and have compressed it as say a frequency table, where 2, 4, 5 are unique values and 3, 2, 2 are frequency of those unique values. Then computing the mean of the vector does not need to scan through the original vector; it is just a weighted mean of the unique values. Of course, the discrete algorithm used for GAM fitting is not as simple as computing mean; but they share the same idea: store data in a compressed manner and do computation using data in this compressed format. This is not only memory efficient, but also computationally cheaper.

While the discrete algorithm is computationally cheapter, it does not practically deliver high performance. Chapter 9 will develop a new computational engine that is truely high performance computing. With such method, GAM fitting for 10^7 or even 10^8 data and 2×10^5 coefficients can be completed in an hour. I call this new computation methods `bamboos`.

It is indeed a great pity that I am unable to build an adequate daily logBS model. In this way I am missing / bypassing many statistical question that are interesting to many statisticians and worth discussing, like preferential sampling. However, I increasingly realize that building model are not where my passion is. I incline to invest more on `bamboos` to make it a standalone computational module / pacakge that can faciliate, not just computations in `mgcv`, but also general GLM fitting.

Chapter 1

Penalized regression splines and additive models

`mgcv` constructs a GAM using low-rank penalized regression splines, which originates from smoothing problems. §1.1 gives an introduction to this, with a focus on the types of splines that are to be applied in this thesis, namely cubic regression splines, thin-plate splines and tensor product splines. Some other aspects of penalized splines are also covered, particularly its link to mixed models and Bayesian inference, which motivates the use of restricted maximum likelihood (REML) for estimating smoothing parameters.

§1.2 moves on to construct a GAM using splines. But since all models in this thesis are additive models, a subclass of GAM, the demonstration will first be restricted to this aspect.

1.1 Smoothing by splines

A smoothing problem is a general regression problem where we want to estimate the underlying data generating process from observed data $(x_1,\ y_1)$, $(x_2,\ y_2)$, ..., $(x_n,\ y_n)$ with i.i.d. Gaussian noise. This can be mathematically formulated as

$$y_i = f(x_i) + \epsilon_i, \quad \epsilon_i \ \sim \ N(0,\ \phi/w_i),$$

where 'N' is Gaussian (or normal) distribution, ϕ is its variance (often denoted by σ^2 in literature elsewhere) and w_i is a known weight for the i^{th} datum. Particularly, the shape of $f(x)$ is not restricted and should be purely "data-driven". If $f(x)$ has a pre-specified parametric representation, like a straight line $f(x) = \beta_0 + \beta_1 x$ or generally a p^{th} order polynomial $f(x) = \sum_{j=0}^{p} \beta_j x^j$, then the problem reduces to a linear regression problem (Faraway, 2004). Well-known estimation methods for $f(x)$ include kernel smoothing (Nadaraya, 1964; Watson, 1964; Wand and Jones, 1994; Bowman and Azzalini, 1997; Hastie et al., 2009), local linear and polynomial regression (Cleveland, 1979; Cleveland and Devlin, 1988; Fan and Gijbels, 1996) and smoothing splines or penalized regression splines to be introduced in this section.

1.1.1 Low-rank penalized cubic regression spline

A spline (de Boor, 1978) is typically a piecewise low-order polynomial that joins smoothly at their connection points, known as *knots*. "Smoothly" means that if the polynomial has order p, then the $(p-1)^{\text{th}}$ derivative of the spline should be continuous at knots. Splines are very useful for

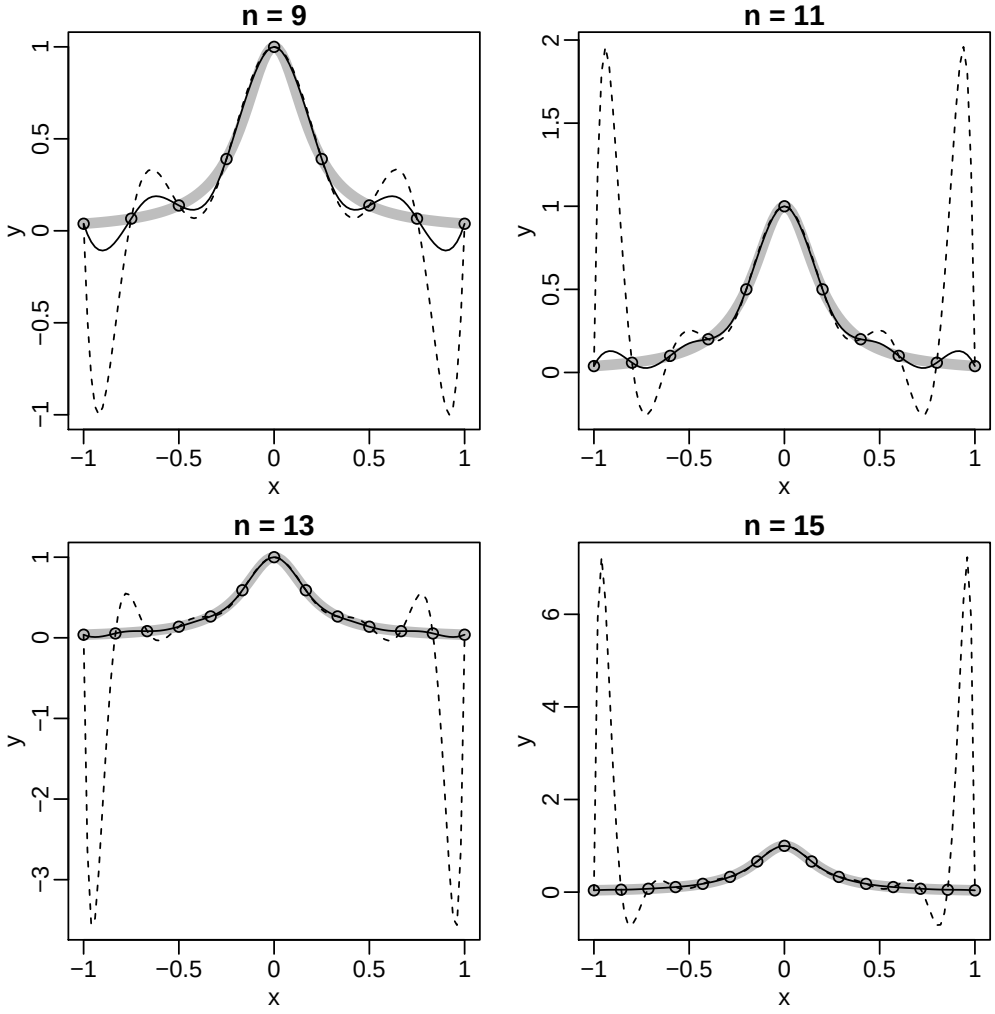


Figure 1.1: Illustration of Runge's phenomenon when using Lagrange polynomial of degree $(n-1)$ to interpolate the (scaled) Runge function $R(x) = 1 \,/\, (1+x^2),\ x \in [-1,\ 1]$ with n evenly spaced sample points. In all graphs, the thick gray curve denotes $R(x)$, the dashed curve denotes Lagrange interpolation polynomial and the dotted curve is a cubic interpolation spline. A higher order polynomial demonstrates increasingly stronger oscillation at two edges as n grows. The approximation polynomial does not uniformly converge to the true function, i.e., a higher order polynomial does not improve approximation. By contrast, an interpolation spline (piecewise cubic polynomials) achieves uniform convergence.

interpolation (Kress, 1996), because they do not suffer from the *Runge's phenomenon* (Epperson, 1987) that high-order polynomial interpolation has. See Figure 1.1 for an illustration.

Splines were first introduced to smoothing problems by Reinsch (1967, 1971), who had a variational problem equivalent to

$$\hat{f}(x) = \underset{f(x)}{\arg\min} \left\{ \sum_{i=1}^{n} w_i \big(y_i - f(x_i)\big)^2 + \lambda \int_{x_1}^{x_n} f''(x)^2 \mathrm{d}x \right\},$$

where λ^{-1} is a Lagrangian parameter. Later in statistical literature, the above is termed a *penalized least squares* problem, and $\int f''(x)^2 \mathrm{d}x$ and $\lambda \geq 0$ are respectively known as a *penalty* of $f(x)$ and a *smoothing parameter* that controls the strength of the penalization. Reinsch showed that the solution $f(x)$ is a cubic spline, and then parametrized it by

$$f(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3, \quad x \in [x_i,\ x_{i+1}),$$

for $i = 1,\ 2,\ \cdots,\ n$ (assuming $x_{n+1} = +\infty$). Under continuity requirement of $f(x)$, $f'(x)$ and $f''(x)$

at all x_i, it can be shown that these parameters are subject to the following constraints:

$$d_i = \frac{h_i}{3}(c_{i+1} - c_i), \qquad b_i = \frac{a_{i+1} - a_i}{h_i} - \Big(\frac{2}{3}c_i + \frac{1}{3}c_{i+1}\Big)h_i,$$
$$\frac{1}{3}h_i c_i + \frac{2}{3}(h_i + h_{i+1})c_{i+1} + \frac{1}{3}h_{i+1}c_{i+2} = \frac{1}{h_i}a_i - \Big(\frac{1}{h_i} + \frac{1}{h_{i+1}}\Big)a_{i+1} + \frac{1}{h_{i+1}}a_{i+2},$$

where $h_i = x_{i+1} - x_i$. In above, there are $4n$ parameters for n data under $3(n-1)$ constraints. Additional boundary constraints, like *natural boundary condition* $c_1 = c_n = d_n = 0$ and *periodic boundary condition* $a_1 = a_n,\ b_1 = b_n,\ c_1 = c_n$ will guarantee a unique solution. Reinsch demonstrated the former, in which case these constraints imply $\boldsymbol{c} = \boldsymbol{T}\boldsymbol{a}$, where $\boldsymbol{T}$ is an $(n-2) \times n$ matrix, $\boldsymbol{c} = (c_2,\ c_3,\ \cdots,\ c_{n-1})'$ and $\boldsymbol{a} = (a_1,\ a_2,\ \cdots,\ a_n)'$. Also note that $f''(x) = 2c_i + \frac{2(c_{i+1}-c_i)}{h_i}(x - x_i)$, so $\int_{x_1}^{x_n} f''(x)^2 \mathrm{d}x = \sum_{i=1}^{n-1} \int_{x_i}^{x_{i+1}} f''(x)^2 \mathrm{d}x = \sum_{i=1}^{n-1} \frac{4}{3}h_i(c_i^2 + c_i c_{i+1} + c_{i+1}^2)$ is a quadratic form of $\boldsymbol{c}$. All together, the penalty can be written in a quadratic form $\boldsymbol{a'Ka}$, where $\boldsymbol{K}$ is an $n \times n$ positive-semidefinite matrix with rank $(n-2)$, called a *penalty matrix*. Particularly, $\boldsymbol{K}$ is a banded matrix with five diagonals, and its elements only depend on h_i. Let $\boldsymbol{y} = (y_1,\ y_2,\ \cdots,\ y_n)'$ and $\boldsymbol{W} = (w_1,\ w_2,\ \cdots,\ w_n)'$, then it is easy to see that the penalized least squares problem becomes

$$\hat{\boldsymbol{a}} = \arg\min_{\boldsymbol{a}} \big\{(\boldsymbol{y} - \boldsymbol{a})'\boldsymbol{W}(\boldsymbol{y} - \boldsymbol{a}) + \lambda \boldsymbol{a'Ka}\big\}.$$

For any known λ, the solution is $\hat{\boldsymbol{a}} = (\boldsymbol{W} + \lambda\boldsymbol{K})^{-1}\boldsymbol{W}\boldsymbol{y}$. Other parameters $\hat{\boldsymbol{b}}$, $\hat{\boldsymbol{c}}$ and $\hat{\boldsymbol{d}}$ can be then determined from their relationship with $\hat{\boldsymbol{a}}$. The resulting spline $\hat{f}(x)$ with estimated parameters is called a *smoothing spline*.

While $f(x)$ is constructed as piecewise polynomials, it can also be written as a linear combination of n basis. To see this, consider an out-of-sample prediction for $f(\tilde{x}_j)$, where $\tilde{x}_j$ is not any of x_i (otherwise the predicted value is just a_i), but still inside $[x_1,\ x_n]$[1]. Define $\boldsymbol{D_j}$ to be an $n \times n$ diagonal matrix with $\boldsymbol{D_j}(i,\ i) = \tilde{x}_j - x_i$, and $\boldsymbol{\delta_j}$ to be a length-n vector with $\boldsymbol{\delta_j}(s_j) = 1$ and all other entries being zeros. The position index s_j satisfies $x_{s_j} \le \tilde{x}_j < x_{s_j+1}$. Then we can write $f(\tilde{x}_j) = \boldsymbol{\delta_j'a} + \boldsymbol{\delta_j'D_j b} + \boldsymbol{\delta_j'D_j^2 c} + \boldsymbol{\delta_j'D_j^3 d}$. The linear constraints between parameters imply that there are linear transformations from $\boldsymbol{b}$, $\boldsymbol{c}$, $\boldsymbol{d}$ to $\boldsymbol{a}$, thus the result can simplify to the following form that only depends on $\boldsymbol{a}$:

$$f(\tilde{x}_j) = \begin{pmatrix}\mathcal{B}_1(\tilde{x}_j) & \mathcal{B}_2(\tilde{x}_j) & \cdots & \mathcal{B}_n(\tilde{x}_j)\end{pmatrix}\boldsymbol{a} = \sum_{i=1}^{n} \mathcal{B}_n(\tilde{x}_j)a_i.$$

To interprete this, replace $\tilde{x}_j$ with x so that $f(x) = \sum_{i=1}^{n} \mathcal{B}_i(x)a_i$. Now it is clear that $\mathcal{B}_i(x)$ is the i^{th} *basis function* and $\mathcal{B}_i(\tilde{x}_j)$ is its value at $x = \tilde{x}_j$. In realistic computation with finite number of data point $\tilde{x}_1$, $\tilde{x}_2$, ..., $\tilde{x}_m$, such basis representation ends up with a matrix equation $\boldsymbol{f} = \boldsymbol{\mathcal{B}a}$, where $\boldsymbol{f} = \big(f(\tilde{x}_1),\ f(\tilde{x}_2),\ \cdots,\ f(\tilde{x}_m)\big)'$ and $\boldsymbol{\mathcal{B}}$ is an $m \times n$ matrix, with $\boldsymbol{\mathcal{B}}(j,\ i) = \mathcal{B}_i(\tilde{x}_j)$. Depending on the use of such matrix, it may be termed differently. For example, in the above out-of-sample prediction problem, it is known as a *predictor matrix*, because it transforms coefficients $\boldsymbol{a}$ to predicted values. If all the basis functions are evaluated at sampling points $x_1,\ x_2,\ \cdots,\ x_n$ instead, the resulting matrix is known as a *design matrix*. Obviously, for this smoothing spline, the design matrix is an identity matrix. It also means that the i^{th} basis function $\mathcal{B}_i(x)$ has value 1 at the i^{th} knot x_i, but value 0 at all other knots. Such basis is known as *cardinal spline basis*.

Thanks to the sparsity of $\boldsymbol{K}$, computation of a smoothing spline is efficient with $O(n)$ complexity. However, at least since Wahba (1980) and Parker and Rice (1985), it has been recognised that using an n dimensional basis representation is computationally wasteful for negligible statistical gain. Gu and Kim (2002), Hall and Opsomer (2005), Li and Ruppert (2008), Kauermann et al. (2009), Claeskens et al. (2009) and Wang et al. (2011) set up a spline with a smaller number of knots, say k, so that it is spanned by k basis: $f(x) = \sum_{i=1}^{k} \mathcal{B}_i(x)a_i$. Such spline is known as a *low-rank penalized regression spline*. They show that k only grows rather slowly with sample size n to achieve asymptotically equivalent results to using a full smoothing spline (e.g. $k = O(n^{1/5})$ for a cubic spline under REML smoothness estimation).

[1] In fact, this is just a cubic spline interpolation problem over $(x_1,\ a_1)$, $(x_2,\ a_2)$, ..., $(x_n,\ a_n)$.

To use a penalized regression spline, one has to place knots, and their locations may not be at known sampling points. So in the basis representation of such a spline, basis coefficients only mean spline's values at knots not at sampling points. Another implication is that the design matrix is no longer an $n \times n$ identity matrix, but an $n \times k$ matrix. The penalty matrix is no longer an $n \times n$ banded matrix, but a $k \times k$ dense matrix. Let us respectively denote this new design matrix and new penalty matrix by $\boldsymbol{X}$ and $\boldsymbol{S}$, the associated penalized least squares problem takes the following form:

$$\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}} = \arg\min_{\boldsymbol{\beta}} \big\{ \|\boldsymbol{W}^{\frac{1}{2}}(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})\|^2 + \lambda \boldsymbol{\beta}'\boldsymbol{S}\boldsymbol{\beta} \big\}.$$

I specially index the estimated coefficients by λ to remind you that it is conditional on given λ. Later in §1.1.6 and §4.3 I will introduce the estimation of λ.

A low-rank cubic regression spline is not a sparse smoother, however. If we represent a spline as a B-spline (de Boor, 1978), whose basis function has *compact support*, we can obtain a sparse design matrix. In penalized regression, a *difference penalty matrix* is often used to preserve such sparsity in estimation. Furthermore, when knots are evenly spaced, such penalty has a simple interpretation as well. This type penalized regression spline is known as *P-splines* (Eilers and Marx, 1996; Marx and Eilers, 1998; Ruppert et al., 2003). Later, Wood (2011, §5.1) developed its adaptive variant, where the smoothing parameter is also represented by a B-spline, so that the strength of the penalization can vary between sampling locations. Recently, Wood (2017) also implemented the conventional derivative-based penalty (like $\int_{x_1}^{x_n} f''(x)^2 \mathrm{d}x$) with B-splines.

1.1.2 Low-rank penalized thin-plate regression spline

A smoothing problem is also defined in $d > 1$ dimensions:

$$y_i = f(x_1,\ x_2,\ \cdots,\ x_d) + \epsilon_i, \quad \epsilon_i \ \sim\ N(0,\ \phi/w_i),$$

where the smooth function f can be represented by a thin-plate spline or other Duchon spline (Duchon, 1977). For example, consider a bivariate example. Given n observations $(x_{11},\ x_{12};\ y_1)$, $(x_{21},\ x_{22};\ y_2)$, ..., $(x_{n1},\ x_{n2};\ y_n)$, a thin-plate spline is the solution to the following penalized least squares problem:

$$f(x_1,\ x_2) = \underset{f(x_1,\ x_2)}{\arg\min} \left\{ \sum_{i=1}^{n} w_i \big(y_i - f(x_{i1},\ x_{i2})\big)^2 + \lambda \iint \Big(\frac{\partial^2 f}{\partial x_1^2}\Big)^2 + 2\Big(\frac{\partial^2 f}{\partial x_1 \partial x_2}\Big)^2 + \Big(\frac{\partial^2 f}{\partial x_2^2}\Big)^2 \mathrm{d}x_1 \mathrm{d}x_2 \right\},$$

and it has a direct basis representation:

$$f(x_1,\ x_2) = \sum_{i=1}^{n} \delta_i \mathcal{B}_i(x_1,\ x_2) + \alpha_0 + \alpha_1 x_1 + \alpha_2 x_2,$$

where $\mathcal{B}_i(x_1,\ x_2) = r_i^2 \log(r_i)$ with Euclidean distance $r_i = \sqrt{(x_1 - x_{i1})^2 + (x_2 - x_{i2})^2}$ is a radial basis centred at a knot $(x_{i1},\ x_{i2})$. Define an $n \times n$ matrix $\boldsymbol{E}$ with $\boldsymbol{E}(j,\ i) = \mathcal{B}_i(x_{j1},\ x_{j2})$ (note that this matrix is symmetric) and an $n \times 3$ matrix $\boldsymbol{T}$ with $\boldsymbol{T}(i,\ ) = (1,\ x_{i1},\ x_{i2})$, then the penalized least squares can be shown to be (Wahba, 1990; Green and Silverman, 1994)

$$(\hat{\boldsymbol{\delta}},\ \hat{\boldsymbol{\alpha}}) = \underset{(\boldsymbol{\delta},\ \boldsymbol{\alpha})}{\arg\min} \big\{ \|\boldsymbol{W}^{\frac{1}{2}}(\boldsymbol{y} - \boldsymbol{E}\boldsymbol{\delta} - \boldsymbol{T}\boldsymbol{\alpha})\|^2 + \lambda \boldsymbol{\delta}'\boldsymbol{E}\boldsymbol{\delta} \big\}, \quad \text{subject to } \ \boldsymbol{T}'\boldsymbol{\delta} = \boldsymbol{0},$$

where $\boldsymbol{y} = (y_1,\ y_2,\ \cdots,\ y_n)'$, $\boldsymbol{\delta} = (\delta_1,\ \delta_2,\ \cdots,\ \delta_n)'$, $\boldsymbol{\alpha} = (\alpha_0,\ \alpha_1,\ \alpha_2)'$.

Since $\boldsymbol{E}$ is a dense matrix, computation for a thin-plate spline involves $O(n^3)$ FLOP which is far too expensive even for moderately large n. Wood (2003) considered an eigen decompostion (see §4.1.5 if you need a revision) of $\boldsymbol{E} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{U}'$, then by retaining the first k eigenvalues $\boldsymbol{D}_{\boldsymbol{k}} = \boldsymbol{D}(1:k,\ 1:k)$ and eigenvectors $\boldsymbol{U}_{\boldsymbol{k}} = \boldsymbol{U}(,\ 1:k)$, a low-rank approximation $\boldsymbol{E}_{\boldsymbol{k}} \approx \boldsymbol{U}_{\boldsymbol{k}}\boldsymbol{D}_{\boldsymbol{k}}\boldsymbol{U}_{\boldsymbol{k}}'$ is obtained. Using $\boldsymbol{E}_{\boldsymbol{k}}$, the penalized least squares problem becomes

$$(\hat{\boldsymbol{\delta}}_{\boldsymbol{k}},\ \hat{\boldsymbol{\alpha}}) = \underset{(\boldsymbol{\delta}_{\boldsymbol{k}},\ \boldsymbol{\alpha})}{\arg\min} \big\{ \|\boldsymbol{W}^{\frac{1}{2}}(\boldsymbol{y} - \boldsymbol{U}_{\boldsymbol{k}}\boldsymbol{D}_{\boldsymbol{k}}\boldsymbol{\delta}_{\boldsymbol{k}} - \boldsymbol{T}\boldsymbol{\alpha})\|^2 + \lambda \boldsymbol{\delta}_{\boldsymbol{k}}'\boldsymbol{D}_{\boldsymbol{k}}\boldsymbol{\delta}_{\boldsymbol{k}} \big\}, \quad \text{subject to } \ (\boldsymbol{U}_{\boldsymbol{k}}'\boldsymbol{T})'\boldsymbol{\delta}_{\boldsymbol{k}} = \boldsymbol{0},$$

where $\boldsymbol{\delta_k} = \boldsymbol{U_k'}\boldsymbol{\delta}$ is a new length-k parameter vector associated with a new $n \times k$ design matrix $\boldsymbol{U_k D_k}$ and a new $k \times k$ diagonal penalty matrix $\boldsymbol{D_k}$. Solving this new problem only requires $O(nk^2)$ FLOP. At first glance, one may wonder why this is benefical. Given that the initial eigen decomposition has $O(n^3)$ (or $O(n^2k)$ if using *Lanczos algorithm*) complexity, the overall complexity for solving this penalized least squares problem is not any lower. However, in practice, the smoothing parameter λ needs be estimated, too. Many trial λ would be attempted, and a penalized least squares problem needs be solved for each trial. Note that the initial eigen decomposition only needs be done once, so all subsequent penalized least squares estimations can be $O(n^2/k^2)$ times faster[2].

For later out-of-sample prediction after estimation, a predictor matrix can be constructed; but to use it, the original parametrization is required. This follows from $\boldsymbol{\delta} = \boldsymbol{U_k}\boldsymbol{\delta_k}$, or $\begin{pmatrix} \boldsymbol{\delta} \\ \boldsymbol{\alpha} \end{pmatrix} = \begin{pmatrix} \boldsymbol{U_k} & \\ & \boldsymbol{I} \end{pmatrix} \begin{pmatrix} \boldsymbol{\delta_k} \\ \boldsymbol{\alpha} \end{pmatrix}$.

1.1.3 Low-rank penalized tensor product regression spline

A thin-plate spline is isotropic due to its representation via radial basis. While this is useful for some settings, like spatial smoothing, it is not in others. Its basic problem is that it is not invariant to linear scaling of variables. For example, if we set $\tilde{x}_2 = 2x_2$ in for the example bivariate thin-plate spline in the previous section, the estimated thin-plate spline will be different.

One way to construct a multivariate spline that is invariant to linear scaling, is via a *tensor product spline*. Such spline can be motivated in several different ways, but the most intuitive illustration may be via a *varying coefficient model*. Consider a bivariate smoothing problem again. We start from an obviously wrong smoothing problem $y_i = g(x_{i1}) + \epsilon$ where we can obtain a cubic spline $\hat{g}(x_1) = \sum_s \mathcal{B}_s(x_1)\alpha_s$. Now to model the effect along the second dimension, we allow the coefficient α_s to vary smoothly with x_2, giving another cubic spline $\alpha_s(t) = \sum_t \mathcal{C}_t(x_2)\beta_{st}$. Note that the new coefficient β_{st} is indexed by two subscripts, because the extension is made for each s. Now if we plug in $\alpha_s(t)$ into $\hat{g}$, we get a new bivariate spline $f(x_1,\ x_2) = \sum_s \sum_t \mathcal{B}_s(x_1)\mathcal{C}_t(x_2)\beta_{st}$. It is clear that the basis for such spline is generated by all pairwise product of two sets of *marginal basis*, or, their *tensor product*, and the construction generalizes to any higher dimension.

When it comes to matrix or vector representation, computation of tensor product basis becomes computation of *Kronecker product*, denoted by $\otimes$. Specifically, the Kronecker product between vectors $\boldsymbol{a} = \begin{pmatrix} a_1 & a_2 & \cdots & a_s \end{pmatrix}$ and $\boldsymbol{b} = \begin{pmatrix} b_1 & b_2 & \cdots & b_t \end{pmatrix}$ is

$$\boldsymbol{a} \otimes \boldsymbol{b} = \begin{pmatrix} a_1\boldsymbol{b} & a_2\boldsymbol{b} & \cdots & a_r\boldsymbol{b} \end{pmatrix},$$

i.e, $\boldsymbol{b}$ is distributed to each element of $\boldsymbol{a}$ for multiplication. The resulting vector has length st. Now, suppose that given n observations, the marginal cubic spline basis for the first and second dimensions respectively end up with a *marginal design matrix* $\boldsymbol{A}$ and $\boldsymbol{B}$ of dimension $n \times t$ and $n \times s$, then if $\boldsymbol{C}$ is the design matrix for the tensor product basis, we have $\boldsymbol{C}(i,\ ) = \boldsymbol{A}(i,\ ) \otimes \boldsymbol{B}(i,\ )$. In other words, we are computing a row-wise Kronecker product between $\boldsymbol{A}$ and $\boldsymbol{B}$, denoted by $\boldsymbol{A} \mathbin{\widetilde{\otimes}} \boldsymbol{B}$ for convenience. The resulting *tensor product design matrix* has dimension $n \times st$.

In practice, a tensor product design matrix is never computed by row, as an alternative definition exists. To see this, consider an example with two 2×2 matrices:

$$\begin{aligned} \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \mathbin{\widetilde{\otimes}} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} &= \begin{pmatrix} a_{11}b_{11} & a_{11}b_{12} & a_{12}b_{11} & a_{12}b_{12} \\ a_{21}b_{21} & a_{21}b_{22} & a_{22}b_{21} & a_{22}b_{22} \end{pmatrix} \\ &= \left(\begin{bmatrix} a_{11} & \\ & a_{21} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \quad \begin{bmatrix} a_{12} & \\ & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \right). \end{aligned}$$

In general, there is

$$\boldsymbol{A} \mathbin{\widetilde{\otimes}} \boldsymbol{B} = \begin{pmatrix} \texttt{diag}(\boldsymbol{A}(,\ 1))\boldsymbol{B} & \texttt{diag}(\boldsymbol{A}(,\ 2))\boldsymbol{B} & \cdots & \texttt{diag}(\boldsymbol{A}(,\ s))\boldsymbol{B} \end{pmatrix}, \tag{1.1}$$

[2]If we further apply the model matrix reduction to be introduced in §4.2, we can further reduce the computational complexity of the penalized least squares problem to $O(k^3)$, hence the estimation process will be $O(n^3/k^3)$ times faster.

$p = \prod_{i=1}^{t} p_{\boldsymbol{B}_i}$
allocate storage for $\boldsymbol{B}$
$\boldsymbol{B} = \text{zeros}(n,\ p)$
copy the last margin into the trailing block of $\boldsymbol{B}$
$q = p_{\boldsymbol{B}_t}$
$\boldsymbol{B}(,\ (p-q+1):p) = \boldsymbol{B_t}$
use backward recursion to fill in all entries of $\boldsymbol{B}$
for $i = (t-1):1$
  $r = q * p_{\boldsymbol{B}_i}$
  $\boldsymbol{B}(,\ (p-r+1):p) = \boldsymbol{B_i} \mathbin{\widetilde{\otimes}} \boldsymbol{B}(,\ (p-q+1):p)$
  $q = r$

Figure 1.2: An efficient algorithm to compute $\boldsymbol{B} = \boldsymbol{B_1} \mathbin{\widetilde{\otimes}} \boldsymbol{B_2} \mathbin{\widetilde{\otimes}} \cdots \mathbin{\widetilde{\otimes}} \boldsymbol{B_t}$.

i.e., $\boldsymbol{B}$ is distributed to each column of $\boldsymbol{A}$ for row-scaling.

Kronecker product follows law of associativity. For three vectors $\boldsymbol{a}$, $\boldsymbol{b}$ and $\boldsymbol{c}$, this implies $\boldsymbol{a} \otimes \boldsymbol{b} \otimes \boldsymbol{c} = (\boldsymbol{a} \otimes \boldsymbol{b}) \otimes \boldsymbol{c} = \boldsymbol{a} \otimes (\boldsymbol{b} \otimes \boldsymbol{c})$. For matrices $\boldsymbol{A}$, $\boldsymbol{B}$ and $\boldsymbol{C}$, there is $\boldsymbol{A} \mathbin{\widetilde{\otimes}} \boldsymbol{B} \mathbin{\widetilde{\otimes}} \boldsymbol{C} = (\boldsymbol{A} \mathbin{\widetilde{\otimes}} \boldsymbol{B}) \mathbin{\widetilde{\otimes}} \boldsymbol{C} = \boldsymbol{A} \mathbin{\widetilde{\otimes}} (\boldsymbol{B} \mathbin{\widetilde{\otimes}} \boldsymbol{C})$. This offers an efficient algorithm for computing $\boldsymbol{B} = \boldsymbol{B_1} \mathbin{\widetilde{\otimes}} \boldsymbol{B_2} \mathbin{\widetilde{\otimes}} \cdots \mathbin{\widetilde{\otimes}} \boldsymbol{B_t}$, as presented in Figure 1.2, where $\boldsymbol{B_i}$ has dimension $n \times p_{\boldsymbol{B}_i}$.

For smoothing purpose, a tensor product spline needs be penalized. Eilers and Marx (2003) used a weighted *Kronecker sum* of *marginal penalty matrices*. For example, if there are three margins, each with a penalty matrix $\boldsymbol{K_i}$, then the tensor product penalty matrix is $\boldsymbol{K} = \lambda_1(\boldsymbol{K_1} \otimes \boldsymbol{I_2} \otimes \boldsymbol{I_3}) + \lambda_2(\boldsymbol{I_1} \otimes \boldsymbol{K_2} \otimes \boldsymbol{I_3}) + \lambda_3(\boldsymbol{I_1} \otimes \boldsymbol{I_2} \otimes \boldsymbol{K_3})$, where the identity matrix $\boldsymbol{I_i}$ has the same dimension with $\boldsymbol{K_i}$. Later Wood (2006) suggested that when possible (for example, if the margins are cubic regression splines not thin-plate regression splines) a reparametrization to each margins will give such penalty matrix a better interpretation. Note that the product here is the usual Kronecker product between matrices. The following is an explanatory example for its computation:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \otimes \boldsymbol{B} = \begin{pmatrix} a_{11}\boldsymbol{B} & a_{12}\boldsymbol{B} \\ a_{21}\boldsymbol{B} & a_{22}\boldsymbol{B} \end{pmatrix}$$

When marginal basis are low-rank spline basis, the resulting tensor product spline basis is also low-rank. Such spline basis is useful for modelling "interaction" between smooth functions. Examples include Belitz and Lang (2008), Augustin et al. (2009), Lee. and Durbán (2011) and Wood et al. (2013).

1.1.4 Canonical form of a low-rank penalized regression spline

Splines introduced so far are those to be applied in this thesis, but there are also other variants of low-rank penalized regression splines. These include soap film smooths (Wood et al., 2008), splines on the sphere (Wahba, 1981) and Gaussian process smoothers (Kammann and Wand, 2003; Handcock et al., 1994), and they are useful in certain spatial applications. A common characteristic of all these low-rank penalized regression splines is that their associated smoothing problem has a canonical form

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\beta}'\boldsymbol{S_\lambda}\boldsymbol{\beta} + \boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim N(\boldsymbol{0},\ \boldsymbol{W}^{-1}\phi), \tag{1.2}$$

where $\boldsymbol{X}$ is a design matrix, $\boldsymbol{\beta}$ is a vector of coefficients, $\boldsymbol{S_\lambda}$ is a penalty matrix (absorbing smoothing parameters), $\boldsymbol{\epsilon} = (\epsilon_1,\ \epsilon_2,\ \cdots,\ \epsilon_n)'$ and $\boldsymbol{W}$ is the inverse correlation matrix with $\boldsymbol{W}(i,\ i) = w_i$ and $\boldsymbol{W}(i,\ j) = 0,\ \forall i \neq j$. The number of rows of $\boldsymbol{X}$ is the number of data. The number of columns of $\boldsymbol{X}$ gives the *rank* of the spline, and $\boldsymbol{X}(,\ i)$ gives the i^{th} spline basis evaluated at sampling points.

Clearly, a cubic regression spline is readily formulated as such. For a tensor product spline, it is also straightforward. $\boldsymbol{X}$ is the tensor product design matrix, and the penalty matrix can be written as

$\boldsymbol{S_\lambda} = \sum_i \lambda_i \boldsymbol{S_i}$, if we for example, define $\boldsymbol{S_1} = \boldsymbol{K_1} \otimes \boldsymbol{I_2} \otimes \boldsymbol{I_3}$ for the three-margin case used in the previous section.

For a thin-plate regression spline, some additional paper work is needed. First, we need to remove the linear constraints. These constraints imply that $\boldsymbol{\delta_k}$ is in the null space of $(\boldsymbol{U_k'T})'$. From §4.1.1, we know that following a thin QR factorization $\boldsymbol{U_k'T} = \boldsymbol{QR}$, $\boldsymbol{Q^\perp}$ is an orthonormal basis for such null space. Thus via another reparametrization $\boldsymbol{\delta_k} = \boldsymbol{Q^\perp \delta_k^*}$, the penalized least squares problem becomes unconstrained in the new parameter space:

$$(\hat{\boldsymbol{\delta}}_{\boldsymbol{k}}^*,\ \hat{\boldsymbol{\alpha}}) = \underset{(\boldsymbol{\delta_k^*},\ \boldsymbol{\alpha})}{\arg\min} \left\{ \|\boldsymbol{W}^{\frac{1}{2}}(\boldsymbol{y} - \boldsymbol{U_k D_k Q^\perp \delta_k^*} - \boldsymbol{T\alpha})\|^2 + \lambda \boldsymbol{\delta_k^{*\prime} Q^{\perp\prime} D_k Q^\perp \delta_k^*} \right\}.$$

Then the result follows if we define $\boldsymbol{X} = \begin{pmatrix} \boldsymbol{U_k D_k Q^\perp} & \boldsymbol{T} \end{pmatrix}$, $\boldsymbol{S_\lambda} = \begin{pmatrix} \lambda \boldsymbol{Q^{\perp\prime} D_k Q^\perp} & \\ & \boldsymbol{0} \end{pmatrix}$ and $\boldsymbol{\beta} = (\boldsymbol{\delta_k^*},\ \boldsymbol{\alpha})'$. Matrix multiplication with $\boldsymbol{Q^\perp}$ only needs three Householder transformations (see §4.1.1 if you need a revision) hence is very efficient.

1.1.5 Bayesian interpretation of smoothing

The penalization on $\boldsymbol{\beta}$ can be interpreted by an (improper) prior $\boldsymbol{\beta} \sim N(\boldsymbol{0},\ \boldsymbol{S_\lambda^{-1}}\phi)$ (Kimeldorf and Wahba, 1970; Wahba, 1983; Silverman, 1985; Fahrmeir and Lang, 2001; Ruppert et al., 2003), where $\boldsymbol{S_\lambda^{-1}}$ is a *Moore-Penrose pseudoinverse* (recalling that $\boldsymbol{S_\lambda}$ is rank-deficient). Then with the basis representation of a penalized regression spline, the canonical smoothing problem (1.2) is just a penalized linear model (or Bayesian linear model, linear mixed model):

$$\boldsymbol{y}|\boldsymbol{\beta} \sim N(\boldsymbol{X\beta},\ \boldsymbol{W^{-1}}\phi), \qquad \boldsymbol{\beta} \sim N(\boldsymbol{0},\ \boldsymbol{S_\lambda^{-1}}\phi). \tag{1.3}$$

Following the basic Bayesian estimation procedure, we write down the (unnormalized) posterior of $\boldsymbol{\beta}$:

$$\mathtt{P}(\boldsymbol{\beta}|\boldsymbol{y}) \propto \mathtt{P}(\boldsymbol{y}|\boldsymbol{\beta})\mathtt{P}(\boldsymbol{\beta}) \propto \exp\left\{ -\frac{\left\|\boldsymbol{W}^{\frac{1}{2}}\boldsymbol{y} - \boldsymbol{W}^{\frac{1}{2}}\boldsymbol{X\beta}\right\|^2 + \boldsymbol{\beta' S_\lambda \beta}}{2\phi} \right\}.$$

For a known λ, the posterior mode is the solution to a *penalized weighted least squares problem*

$$\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}} = \underset{\boldsymbol{\beta}}{\arg\min} \left\|\boldsymbol{W}^{\frac{1}{2}}\boldsymbol{y} - \boldsymbol{W}^{\frac{1}{2}}\boldsymbol{X\beta}\right\|^2 + \boldsymbol{\beta' S_\lambda \beta}. \tag{1.4}$$

To find its solution, define the *weighted residual sum of squares* and the *penalized weighted residual sum of squares* by

$$D(\boldsymbol{\beta}) = \|\boldsymbol{W}^{\frac{1}{2}}\boldsymbol{y} - \boldsymbol{W}^{\frac{1}{2}}\boldsymbol{X\beta}\|^2, \qquad D_p(\boldsymbol{\beta},\ \boldsymbol{\lambda}) = D(\boldsymbol{\beta}) + \boldsymbol{\beta' S_\lambda \beta}, \tag{1.5}$$

then solve $\partial D_p / \partial \boldsymbol{\beta} = \boldsymbol{0}$. It is easy to derive that

$$\begin{aligned} \frac{\partial D}{\partial \boldsymbol{\beta}} &= -2\boldsymbol{X'W}(\boldsymbol{y} - \boldsymbol{X\beta}), & \frac{\partial^2 D}{\partial \boldsymbol{\beta} \partial \boldsymbol{\beta'}} &= 2\boldsymbol{X'WX}, \\ \frac{\partial D_p}{\partial \boldsymbol{\beta}} &= -2(\boldsymbol{X'Wy} + \boldsymbol{H_\lambda \beta}), & \frac{\partial^2 D_p}{\partial \boldsymbol{\beta} \partial \boldsymbol{\beta'}} &= 2\boldsymbol{H_\lambda}, \end{aligned} \tag{1.6}$$

where

$$\boldsymbol{H_\lambda} = \boldsymbol{X'WX} + \boldsymbol{S_\lambda}. \tag{1.7}$$

Thus the solution to (1.4) is

$$\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}} = \boldsymbol{H_\lambda^{-1} X'Wy}. \tag{1.8}$$

In fact, the posterior is a Gaussian distribution (because the exponent is a quadratic form of $\boldsymbol{\beta}$), so this posterior mode is also the posterior mean. Furthermore, the posterior variance is $\boldsymbol{H_\lambda^{-1}}\phi$, hence altogether, there is

$$\boldsymbol{\beta}|\boldsymbol{y} \sim N\big(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{H_\lambda^{-1}}\phi\big).$$

Let $\hat{\boldsymbol{y}} = \boldsymbol{X}\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}$ be fitted values and $\hat{\boldsymbol{z}} = \boldsymbol{Z}\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}$ be some out-of-sample predicted values with predictor matrix $\boldsymbol{Z}$, then it is straightforward to see that they respectively have variance $\boldsymbol{X}\boldsymbol{H}_{\boldsymbol{\lambda}}^{-1}\boldsymbol{X}'\phi$ and $\boldsymbol{Z}\boldsymbol{H}_{\boldsymbol{\lambda}}^{-1}\boldsymbol{Z}'\phi$. A convenient estimate (although not unbiased) of ϕ is $\hat{\phi} = D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda})/\big(n - \mathtt{tr}(\boldsymbol{A}_{\boldsymbol{\lambda}})\big)$, where $\boldsymbol{A}_{\boldsymbol{\lambda}} = \boldsymbol{X}\boldsymbol{H}_{\boldsymbol{\lambda}}^{-1}\boldsymbol{X}'\boldsymbol{W}$ is a *hat matrix* or an *influence matrix* (so that $\hat{\boldsymbol{y}} = \boldsymbol{A}_{\boldsymbol{\lambda}}\boldsymbol{y}$), and $\mathtt{tr}(\boldsymbol{A}_{\boldsymbol{\lambda}})$ is one definition of *effective degrees of freedom* for the estimated spline.

The above idea of estimation belongs to *empirical Bayesian inference*, not *hierarchical Bayesian inference* (or full Bayesian inference), as no prior distributions are assumed for λ and ϕ. However, given the link between smoothing and mixed modelling, several work (Zuur et al., 2014; Crainiceanu et al., 2005; Wood, 2016) has implemented full Bayesian inference via BUGS (Spiegelhalter et al., 1996) or JAGS (Plummer, 2003).

1.1.6 Smoothness selection via REML

Generally the smoothing parameter $\boldsymbol{\lambda}$ needs be estimated. A conventional criterion is to choose one that minimizes the *generalized cross validation (GCV)* score (Craven and Wahba, 1979; Wood, 2004, 2008)

$$\mathrm{GCV}(\boldsymbol{\lambda}) = \frac{D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda})}{\big(1 - \mathtt{tr}(\boldsymbol{A}_{\boldsymbol{\lambda}})/n\big)^2}.$$

However, Reiss and Ogden (2009) showed that a *restricted likelihood* (Patterson and Thompson, 1971) approach offers practical reliability advantages, being less prone to multiple local optima and consequent under-smoothing.

For the smoothing problem (1.2), the log restricted likelihood is

$$l_r(\boldsymbol{\lambda},\ \phi) = \log \int \mathtt{P}(\boldsymbol{y}\,|\,\boldsymbol{\beta})\mathtt{P}(\boldsymbol{\beta})\,\mathrm{d}\boldsymbol{\beta}.$$

The best estimate of $\boldsymbol{\lambda}$ and ϕ maximizes $l_r(\boldsymbol{\lambda},\ \phi)$, or minimizes a *REML score* $\mathcal{V}_r(\boldsymbol{\lambda},\ \phi) = -2l_r(\boldsymbol{\lambda},\ \phi)$ (Wood, 2011; Wood et al., 2015). The integral in $l_r(\boldsymbol{\lambda},\ \phi)$ has a closed form (as the integrand is the unnormalized posterior of $\boldsymbol{\beta}$ which is Gaussian). Let m be the null space dimension of $\boldsymbol{S}_{\boldsymbol{\lambda}}$ (or the number of $\boldsymbol{S}_{\boldsymbol{\lambda}}$'s zero eigenvalues) and $|\boldsymbol{S}_{\boldsymbol{\lambda}}|_+$ be a generalized determinant (or the product of $\boldsymbol{S}_{\boldsymbol{\lambda}}$'s non-zero eigenvalues), then there are

$$\mathtt{P}(\boldsymbol{y}\,|\,\boldsymbol{\beta}) = (2\pi\phi)^{-\frac{n}{2}}|\boldsymbol{W}|^{\frac{1}{2}}\exp\{-\tfrac{D(\boldsymbol{\beta})}{2\phi}\},\quad \mathtt{P}(\boldsymbol{\beta}) = (2\pi\phi)^{-\frac{p-m}{2}}|\boldsymbol{S}_{\boldsymbol{\lambda}}|_+^{\frac{1}{2}}\exp\{-\tfrac{\boldsymbol{\beta}'\boldsymbol{S}_{\boldsymbol{\lambda}}\boldsymbol{\beta}}{2\phi}\},$$
$$\mathtt{P}(\boldsymbol{y}\,|\,\boldsymbol{\beta})\mathtt{P}(\boldsymbol{\beta}) = (2\pi\phi)^{-\frac{n+p-m}{2}}|\boldsymbol{W}|^{\frac{1}{2}}|\boldsymbol{S}_{\boldsymbol{\lambda}}|_+^{\frac{1}{2}}\exp\{-\tfrac{D_p(\boldsymbol{\beta},\ \boldsymbol{\lambda})}{2\phi}\}.$$

Following an exact Taylor expansion of $D_p(\boldsymbol{\beta},\ \boldsymbol{\lambda})$ at its minimum point $\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}$ (1^{st} derivative at $\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}$ is zero; see (1.6) for 2^{nd} derivative):

$$D_p(\boldsymbol{\beta},\ \boldsymbol{\lambda}) = D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda}) + (\boldsymbol{\beta} - \hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}})'\boldsymbol{H}_{\boldsymbol{\lambda}}(\boldsymbol{\beta} - \hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}),$$

the REML score can be shown to be

$$\mathcal{V}_r(\boldsymbol{\lambda},\ \phi) = (n-m)\log(2\pi\phi) + \frac{D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda})}{\phi} + \log|\boldsymbol{H}_{\boldsymbol{\lambda}}| - \log|\boldsymbol{S}_{\boldsymbol{\lambda}}|_+ - \log|\boldsymbol{W}|. \tag{1.9}$$

The minimizer of $\mathcal{V}_r$ needs be found numerically as there isn't an analytical solution. Later in §4.3 I will demonstrate the Newton-Raphson method for this minimization task in great details.

1.2 Additive models via penalized regression splines

An *additive model* is an extension of a smoothing problem. It has a structure like

$$y_i = f_1(x_i) + f_2(z_i) + f_3(u_{i1},\ u_{i2}) + f_4(v_{i1},\ v_{i2},\ v_{i3}) + \cdots + \epsilon_i, \quad \epsilon_i \ \sim \ N(0,\ \phi/w_i),$$

where the observed data are modelled by a number of additive smooth functions and f_i is the i^{th} component. An additive model is a special case of a *generalized additive model (GAM)* (Hastie and Tibshirani, 1986, 1990) (described later in §4.4). In full generality, a combination of different smooth functions can be used for model representation. For example, f_1 can be a kernel smoother, f_2 can be a loess line, f_3 can be a thin-plate spline, and f_4 can be a tensor product spline with all margins being cubic splines. A convenient estimation method for such a model is via a *backing fitting* algorithm, implemented as an iterative, component-wise smoothing problem. Another appealing representation is to express all smooth functions by low-rank penalized regression splines (Wahba, 1990). Since all splines have a basis representation and a quadratic penalty, they can then be combined into a "larger" penalized regression problem. In this way, smoothness selection be carried out by minimizing GCV or REML. In addition, parametric effects and random effects can also be easily integrated into the model structure and be estimated.

Here is a simple, illustrative example for an additive model via penalized regression spline:

$$y_i = \alpha_0 + u_i\alpha_1 + v_i\alpha_2 + f_1(x_i) + f_2(z_{i1},\ z_{i2}) + \epsilon_i, \quad \epsilon_i \ \sim \ N(0,\ \phi/w_i),$$

where $\alpha_0 + u_i\alpha_1 + v_i\alpha_2$ is a parametric component via a linear regression on variables u and v, and f_1 and f_2 are some smooth functions represented with low-rank penalized regression splines. We assume that $\boldsymbol{f_1} = \boldsymbol{X_1}\boldsymbol{\beta_1}$ is a spline with a single penalty matrix $\lambda_1\boldsymbol{S_1}$, and $\boldsymbol{f_2} = \boldsymbol{X_2}\boldsymbol{\beta_2}$ is a spline with a total penalty matrix from q_2 additive penalty matrices $\boldsymbol{S_2} = \sum_{i=1}^{q_2}\lambda_{2i}\boldsymbol{S_{2i}}$. Following our previous introductions on penalized regression spline, we see that f_1 may be a cubic spline, and f_2 may be a tensor product spline. However, there could be other possibilies that I have mentioned. For example, f_1 may also be a univariate thin-plate spline or a P-spline, and f_2 may be an adaptive P-spline or any type of splines with additional penalization on its null space (Wood, 2011, §5.3). Given such setup, the additive model can then be written as

$$\boldsymbol{y} = \boldsymbol{X_0}\boldsymbol{\beta_0} + \boldsymbol{X_1}\boldsymbol{\beta_1} + \boldsymbol{X_2}\boldsymbol{\beta_2} + \lambda_1\boldsymbol{\beta_1'}\boldsymbol{S_1}\boldsymbol{\beta_1} + \boldsymbol{\beta_2'}\boldsymbol{S_2}\boldsymbol{\beta_2} + \boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \ \sim \ N(\boldsymbol{0},\ \boldsymbol{W}^{-1}\phi), \tag{1.10}$$

where

$$\boldsymbol{X_0} = \begin{pmatrix} 1 & u_1 & v_1 \\ 1 & u_2 & v_2 \\ \vdots & \vdots & \vdots \\ 1 & u_n & v_n \end{pmatrix}, \quad \boldsymbol{\beta_0} = \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \end{pmatrix}$$

are the design matrix and the coefficient vector for the parametric component. By further defining

$$\boldsymbol{X} = \begin{pmatrix} \boldsymbol{X_0} & \boldsymbol{X_1} & \boldsymbol{X_2} \end{pmatrix}, \quad \boldsymbol{\beta} = \begin{pmatrix} \boldsymbol{\beta_0} \\ \boldsymbol{\beta_1} \\ \boldsymbol{\beta_2} \end{pmatrix}, \quad \boldsymbol{S_\lambda} = \begin{pmatrix} \boldsymbol{0} & & \\ & \lambda_1\boldsymbol{S_1} & \\ & & \sum_{i=1}^{q_2}\lambda_{2i}\boldsymbol{S_{2i}} \end{pmatrix}, \tag{1.11}$$

the additive model (1.10) becomes a canonical smoothing model (1.2) with Bayesian interpretation (1.3) and penalized weighted least squares problem (1.4). Its GCV and REML scores also follow those for the smoothing model. For example, let m be the null space dimension of $\boldsymbol{S_\lambda}$ (or the number of $\boldsymbol{S_\lambda}$'s zero eigenvalues), the REML score is just (1.9).

Chapter 2

Additive models with splines in practice: simulated examples

In this Chapter I will use two simulated examples to briefly demonstrate some practical aspects of statistical modelling via smoothing or additive models with splines. These include

- How to properly model data with a cyclic characteristic?
- How to place knots and choose appropriate number of knots for a spline?
- How to reasonably do exploratory data analysis on data with autocorrelation?
- How to estimate an additive model with AR(1) model error?

Simulated datasets are useful as they help assess how well a method work. In addition, I will cover some visualization methods for model inspection and checking. This Chapter is a "warm-up" for the the next Chapter on statistical modelling of real-world data.

2.1 Classic example with independent errors

Consider a signal function (i.e., the "truth")

$$h(x) = 2\pi x \sin(2\pi x) e^{-\frac{2\pi x}{5}}, \quad x \in [0,\ 4].$$

This function factorizes into two parts: $A(x) = 2\pi x e^{-\frac{2\pi x}{5}}$ and $B(x) = \sin(2\pi x)$. While $h(x)$ is not a period function, it inherits the cyclic characteristic of $B(x)$ and crosses zero at all whole numbers. 801 evenly spaced samples are taken on $[0,\ 1]$: $x_i = 0.005(i-1)$, $i = 1,\ 2,\ \cdots,\ 801$, and assume that samples of the signal are observed with i.i.d. Gaussian noise

$$y_i = h(x_i) + \epsilon_i, \quad \epsilon_i \sim N(0,\ \phi).$$

I will choose $\phi = 0.09 \times \texttt{var}(h(x_i))$ so that the noise-to-signal ratio is 0.3. Panel (a) of Figure 2.1 sketches a random set of observed data (see Figure caption for the `R` code to reproduce them). Pretending that we don't know the mathematical form of $h(x)$, we want to estimate it from the data using splines.

It is straightforward to fit a smoothing spline or a penalized regression spline on this dataset (see panel (b) of the Figure) and the fitted spline looks good, but it is undesired if we want to properly

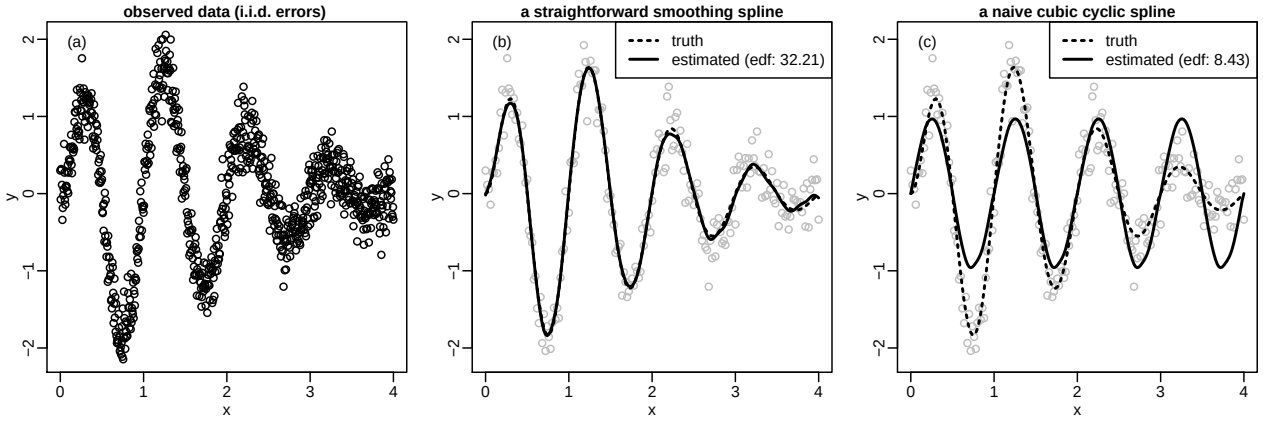


Figure 2.1: Example data used for §2.1. The `R` code to reproduce them is `set.seed(0); x <- seq(0, 4, 0.005); A <- 2 * pi * x * exp(-2 * pi * x / 5); fx <- A * sin(2 * pi * x); y <- fx + rnorm(801, 0, 0.3 * sd(fx)); dat <- data.frame(x = x, A = A, fx = fx, y = y)`. Panel (a) is created by `with(dat, plot(x, y))`. The smoothing spline in panel (b) is obtained by `sm <- smooth.spline(dat$x, dat$y)`, where 140 knots (`sm$fit$nk - 2`) are placed by default. The resulting smoothing spline has degree of freedom 32.21 (obtained from `sm$df`). The simple cubic cyclic spline is estimated in `mgcv`, with `cycl <- gam(y ~ s(x, bs = "cc", k = 16), data = dat, knots = list(x = c(0, 1), method = "REML"))`. The effective degree of freedom is 8.43 (obtained from `sum(cycl$edf)`). `gam` is used here because using `bam` for such a small dataset is an overkill. Note that in panel (c) and (d), as well as all subsequent Figures associated with this example, only every 4 data are plotted (in gray dots), i.e., at $i = 1,\ 5,\ 9, \cdots,\ 801$. This is for a clearer display of the true signal (dashed line) and the estimated spline (solid line).

model the cyclic effect (I will elaborate on this later in §2.1.2). By contrast, the cyclic effect can be easily captured by using a cubic cyclic spline (a variant of cubic regression spline introduced in §1.1.1, by modifying the natural boundary condition to the periodic condition where the function values equal at two ends), but the estimation result (see panel (c) of Figure 2.1) is not satisfying as it fails to model the variability in the data as x changes. Can we add more components onto the cyclic spline to get a more appropriate fit?

2.1.1 Tensor product spline for interaction

It is not obvious how this can be done. The classic "trend + seasonality" decomposition does not help here as the data clearly have no trend, leaving the "seasonality" as the only component.

To get some inspiration let us do some cheating. We know that $h(x_i) = A(x_i)B(x_i)$ in the true signal. First of all, if we know every $A_i = A(x_i)$, estimation of $B(x)$ should be very accurate from the following model

$$y_i = A_i f_0(x_i) + \epsilon_i,$$

where $f_0(x)$ is a cubic cyclic spline to estimate $B(x)$. Note that the presence of A_i does not make model estimation any more difficult. By expressing the model in matrix form

$$\boldsymbol{y} = \texttt{diag}(\boldsymbol{A})\boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon},$$

where $\boldsymbol{A} = (A_1,\ A_2, \cdots,\ A_n)$ collects all A_i's into a vector, we see that estimation of $\boldsymbol{\beta}$ is just a regression problem with design matrix $\texttt{diag}(\boldsymbol{A})\boldsymbol{X}$.

Now what can we do if A_i is unknown? Can we approximate it with a polynomial of x_i, giving an approximation function

$$\left(\sum_{k=0}^{p} x_i^k \alpha_k\right) f_0(x_i) = f_0(x_i) + \sum_{k=1}^{p} x_i^k \alpha_k f_1(x_i) = f_0(x_i) + \sum_{k=1}^{p} x_i^k f_k(x_i),$$

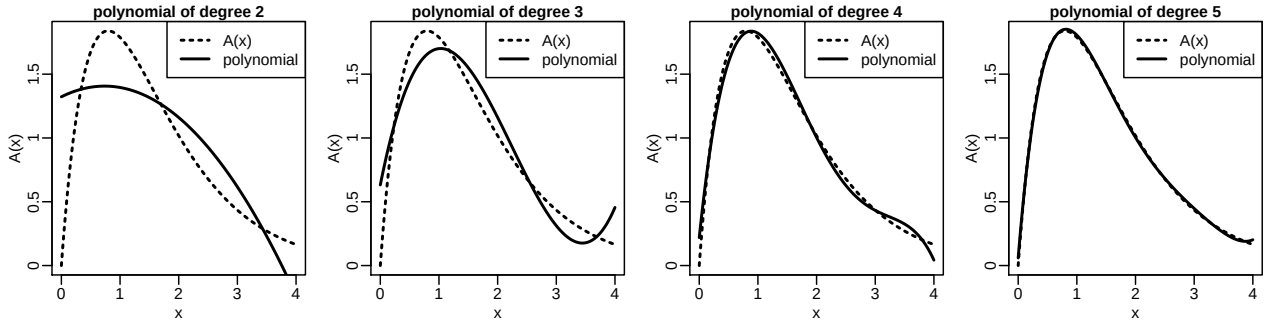


Figure 2.2: Polynomial approximation to $A(x)$. The approximation is good enough when degree reaches 5. The regression polynomials are obtained via simple least squares fitting with function `lm`. For example, the cubic polynomial is obtained by `lm(A ~ poly(x, degree = 3), data = dat)`.

where the unknown polynomial coefficient α_k is absorbed into the unknown cyclic spline $f_0(x_i)$ for a new unknown cyclic spline $f_k(x_i)$? As Figure 2.2 shows, this seems a reasonable idea. A polynomial of degree 5 is a very good approximation to $A(x)$.

We can also approximate A_i with a cubic regression spline $\sum_s \mathcal{B}_s(x_i)\alpha_s$ rather than a high order polynomial (as said at the beginning of §1.1.1, this is desirable). If we express $f_0(x_i) = \sum_t \mathcal{C}_t(x_i)\beta_t$, we see that the approximation function

$$\left(\sum_s \mathcal{B}_s(x_i)\alpha_s\right)\left(\sum_t \mathcal{C}_t(x_i)\beta_t\right) = \sum_t\sum_s \mathcal{B}_s(x_i)\mathcal{C}_t(x_i)\alpha_s\beta_t := \sum_t\sum_s \mathcal{B}_s(x_i)\mathcal{C}_t(x_i)\gamma_{st}$$

is a tensor product spline, where one margin is a natural cubic spline of x and the other is a cubic cyclic spline of x.

You may have been confused why a tensor product spline, a multivariate spline, turns up in this univariate context. To clarify this, let us use variable x for the natural cubic spline and variable $z = x - \lfloor x \rfloor$ for the cyclic cubic spline, where $\lfloor x \rfloor$ is the biggest integer no larger than x. Note that $x \in [0,\ 4]$, but $z \in [0,\ 1]$ (where a period of the cyclic spline is defined). Then we can express the true signal $h(x)$ as a bivariate function $h(x, z) = A(x)B(z)$. From now on, I will use this bivariate representation for the true signal and approximation functions.

In summary, we have come up with the following candidate models to estimate $h(x, z)$:

Model 2.1: "true" model

$$y_i = A_i f_0(z_i) + \epsilon_i,$$

Model 2.2: polynomial model

$$y_i = f_0(z_i) + \sum_{k=1}^{p} x_i^k f_k(z_i) + \epsilon_i,$$

Model 2.3: tensor product spline

$$y_i = f_0(z_i) + f_1(z_i,\ x_i) + \epsilon_i.$$

I would like to emphasize that the tensor product spline $f_1(z_i,\ x_i)$ should not be expressed in a separable form like $g_1(z_i)g_2(x_i)$, even if it has g_1 and g_2 as its margins. The separable form is a common misconception. In fact, a tensor product spline is much more flexible that the mere spline product. Suppose g_1 and g_2 respectively have k_1 and k_2 spline coefficients, the tensor product spline f_1 will have k_1k_2 spline coefficients, while the spline product g_1g_2 only has $k_1 + k_2$ spline coefficients.

All three models can be easily fitted with `R` package `mgcv`, using the REML estimation method introduced in §4.3. Both natural cubic splines and cubic cyclic splines are knots based splines so

we have to supply knots so that spline basis can be constructed and evaluated to produce a design matrix. I will discuss knots related issues in the coming section, §2.1.3. For now, it is sufficient to know that setting $10 \sim 16$ equally spaced knots for both x and z in their range are more than sufficient.

Figure 2.3 illustrates fitted splines for all these models. The best fit is surely model 2.1 (see panel (a)). No approximation for $A(x)$ is done, so the cyclic spline can be accurately estimated. Nevertheless, this model is "cheating" as we should pretend that we don't know $A(x)$. Panels (b) to (e) are model 2.2 with polynomials of degree 2, 3, 4 and 5. Although it appears from Figure 2.2 that polynomial approximation to $A(x)$ is only good enough when degree reaches 5, estimation for $h(x, z)$ is readily good enough when a cubic polynomial is used. Nevertheless, model 2.2 evidently overfits the data. Panel (f) sketches the fitted model 2.3. While there is also a little overfitting near ther right boundary, it is less pronounced than that in the polynomial model.

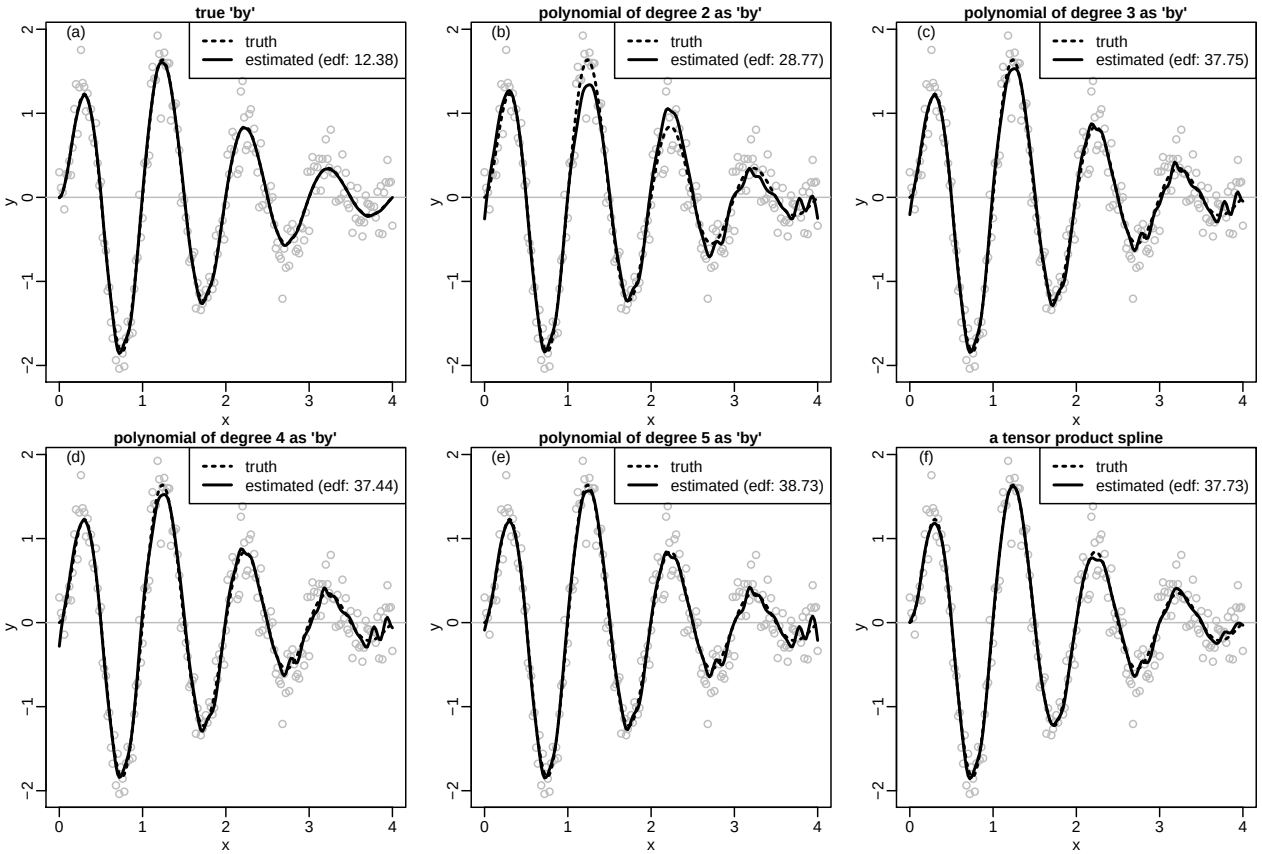


Figure 2.3: Fitting models 2.1, 2.2 and 2.3 in `mgcv`. First create variable z: `dat$z <- dat$x - floor(dat$x)`. For the fitted spline in panel (a), use `gam(y ~ s(z, bs = "cc", k = 16, by = A), data = dat, knots = list(z = c(0, 1)), method = "REML")`. For the fitted spline in panel (c) for example, use `gam(y ~ s(z, bs = "cc", k = 16) + s(z, bs = "cc", k = 16, by = x) + s(z, bs = "cc", k = 16, by = I(x*x)), data = dat, knots = list(z = c(0, 1)), method = "REML")`. For the fitted tensor product spline in panel (f), use `gam(y ~ s(z, bs = "cc", k = 16) + ti(x, z, bs = c("cr", "cc"), k = c(12, 10)), data = dat, knots = list(z = c(0, 1)), method = "REML")`. Note the use of `ti` not `te` for properly creating interaction.

There is yet another reason why using polynomials should not be recommended, as is illustrated in Figure 2.4. The Figure plots every additive component in the model when the polynomial has degree 3. In other words, the fitted spline in panel (c) of Figure 2.3 are decomposed as (a) + (b) + (c2) + (d2) + (e2) of this Figure. It is concerning that terms in (c2), (d2) and (e2) cancel out each other. For example, $xf_1(z)$ is positive in (roughly) $[3,\ 3.7]$, but $x^2 f_2(z)$ is negative in that range. This is an alternative illustration of Runge's phenomenon (previously mentioned in §1.1.1), i.e., the oscillation of high order polynomials. In practice, such "cancellation" lacks numerical stability as many significant digits may be lost in the way, giving inaccurate computation result in finite-precision floating-point arithmetic.

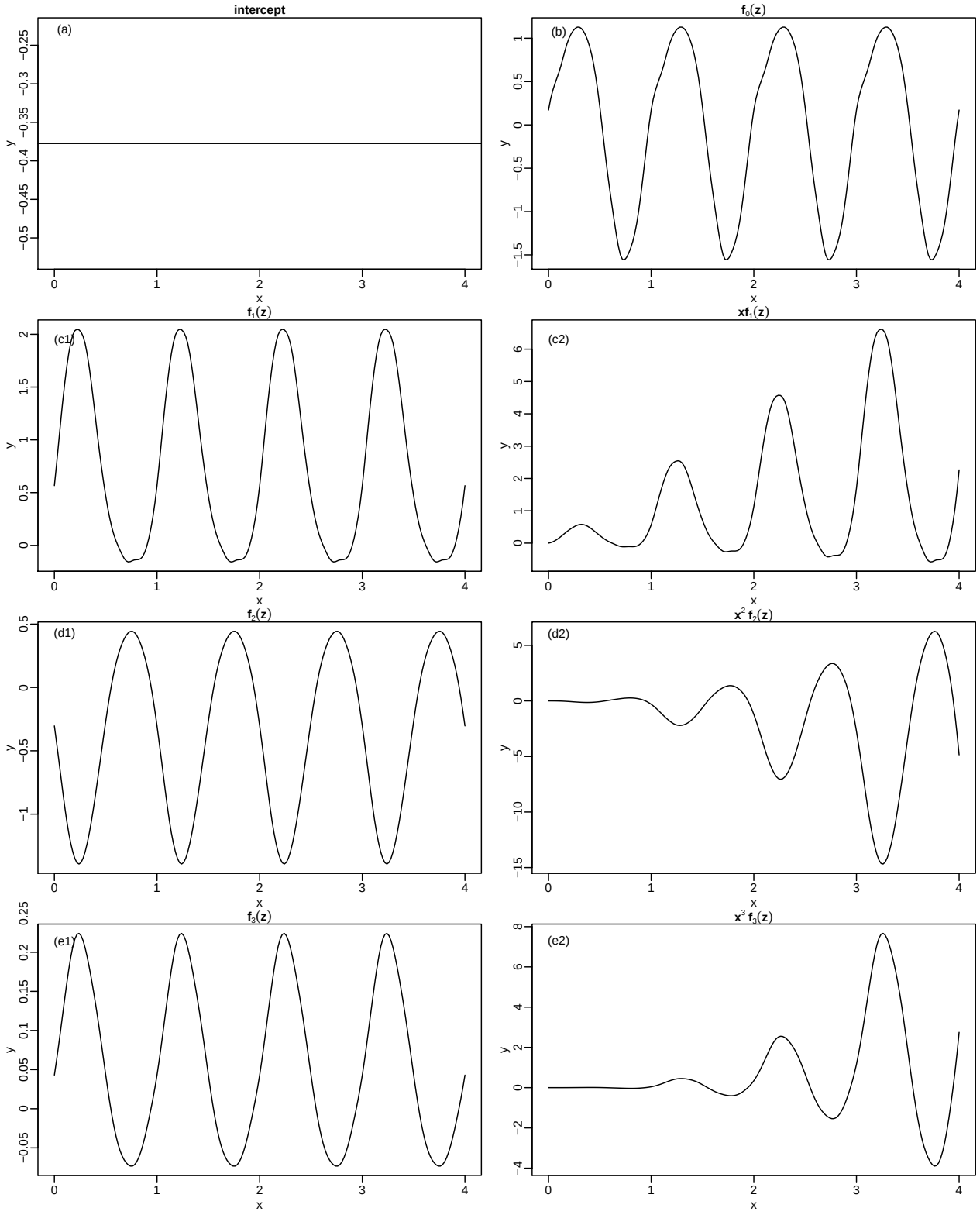


Figure 2.4: Each additive component of model 2.2. Panel (a) is the intercept; Panel (b) is $\hat{f}_0$ (the estimated function for f_0). Panel (c1), (d1) and (e1) are $\hat{f}_1$, $\hat{f}_2$ and $\hat{f}_3$; they are multiplied respectively by x, x^2 and x^3 when they enter the model, which are sketched in panel (c2), (d2) and (e2). It is concerning that terms in (c2), (d2) and (e2) cancel out each other, as they have different sign in the same range. Such "cancellation" makes finite-precision numerical computation less accurate.

2.1.2 Why is an additive model better than a simple smoothing spline?

I have previously mentioned that using a smoothing spline to model $h(x, z)$ is not a good idea; here is the reason. Suppose now our samples are not even spaced over $[0,\ 4]$. For simplicity let us assume that some of the original evenly spaced data are missing. Let us examine two cases:

1. Data near two boundaries are missing: observations indexed by $i = 1,\ 2, \cdots,\ 100$ and $i = 702,\ 703,\ \cdots,\ 801$ are missing;
2. Data in the middle are missing: observations indexed by $i = 201,\ 201,\ \cdots,\ 400$ are missing.

For each case we fit a simple smoothing spline and a tensor product model 2.3 and make prediction at those unsampled positions. The results are illustrated in Figure 2.5.

1. In the first case, prediction is an extrapolation problem. The natural boundary condition assumed by a smoothing spline leads to linear extrapolation (see panel (1a)). The tensor product spline successfully preserves the cyclic nature of the singal, although it overestimates the amplitude at the left boundary.
2. In the second case, prediction is an interpolation problem. The smoothing spline (see panel (2a)) underestimates the true function while the tensor product spline (see panel (2b)) yields good approximation to the truth.

2.1.3 Knots placement and choosing number of knots

In the previous section various splines have been fitted to the example data, but I haven't explained where, how and how many knots are placed for construction of these splines.

In general, suppose we want to place k knots $g_1,\ g_2, \cdots,\ g_k$ in order to estimate a regression spline $y_i = f(x_i) + \epsilon_i$, the basic requirements on knots placement are

1. At most one knot exist between any two adjacent unique data values x_i and x_{i+1};
2. At least one unique data value exists between any two adjacent knots g_j and g_{j+1}

These two statements are in fact equivalent. Violation of either will lead to more than one knot between two two unique data values, which is problematic. For example, suppose there are two knots g_j and g_{j+1} between data values x_i and x_{i+1}, then the piecewise polynomial on $[g_j,\ g_{j+1}]$ can not be uniquely determined as there are no data on this interval.

There is no unique way for knots placement, however, a canonical and automatic knots placement method is to place knots by quantiles of unique data values. This will meet both requirements, and achieve a stronger result on the second: there will be roughly equal number of unique data values on any $[g_j,\ g_{j+1}]$. See Figure 2.6 for an illustration.

If knots are automatically placed as above, knots placement is only up to choosing the number of knots. Choosing an appropriate number of knots is important for a regression spline, since the number of knots, k, decides the number of coefficients to parametrize a spline: the larger k is, the more flexible the spline is hence the closer it can approximate the data. When k is insufficient big, the spline can

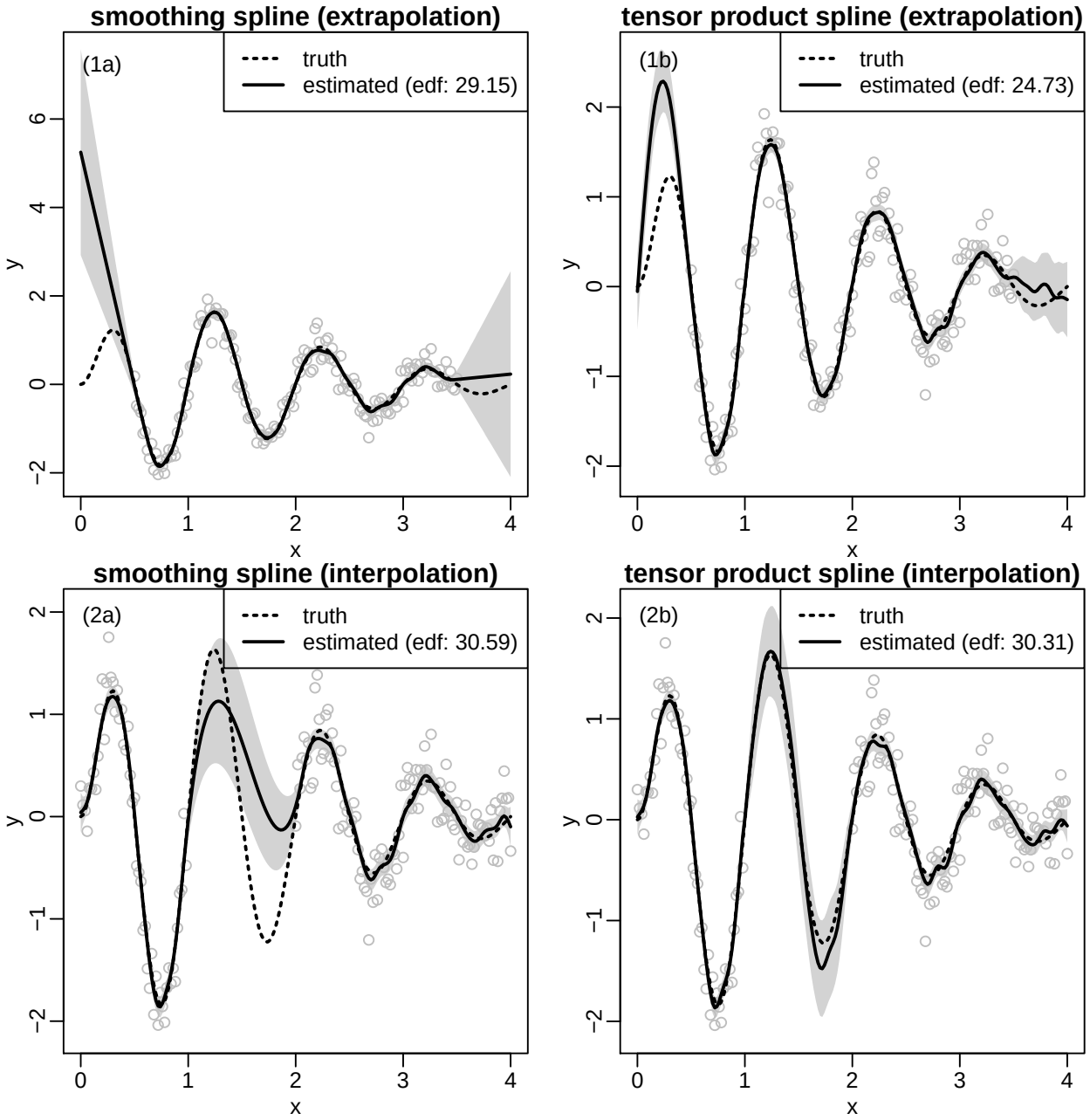


Figure 2.5: Fitting a smoothing spline and a tensor product model 2.3 to non-evenly spaced samples on [0, 4]. In panel (1a) and (1b), data are missing near the two boundaries: `subset1 <- dat[101:701, ]`. The smoothing spline in panel (1a) is estimated by `gam(y ~ s(x, bs = "cr", k = 50), data = subset1, method = "REML")` and the tensor product spline in panel (1b) is estimated by `gam(y ~ s(xt, bs = "cc", k = 16) + ti(xt, x, bs = c("cc", "cr"), k = c(12, 10)), data = subset1, method = "REML")`. In panel (2a) and (2b), data are missing in the middle: `subset2 <- dat[c(1:200, 401:801), ]`. The smoothing spline in panel (2a) is estimated by `gam(y ~ s(xt, bs = "cc", k = 16) + ti(xt, x, bs = c("cc", "cr"), k = c(12, 10)), data = subset2, method = "REML")` and the the tensor product spline in panel (2b) is estimated by `gam(y ~ s(xt, bs = "cc", k = 16) + ti(xt, x, bs = c("cc", "cr"), k = c(12, 10)), data = subset2, method = "REML")`. 95% Bayesian confidence intervals (gray shaded) are produced by `plot.gam`. In both extrapolation and interpolation scenarios, the smoothing spline fails to well model the cyclic nature of the true signal.

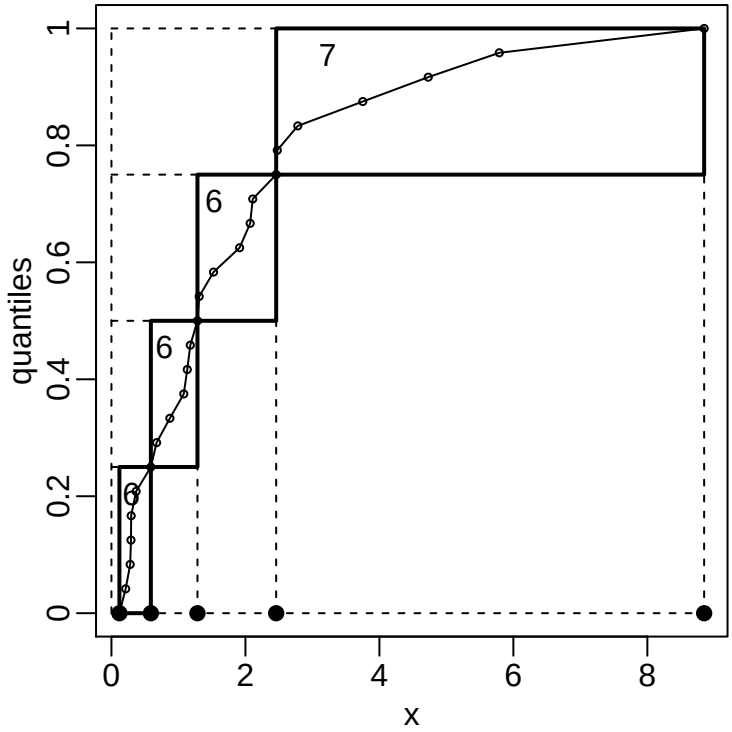


Figure 2.6: Automatic knots placement by quantiles of unique data values. Suppose we have 25 unique x-values: `set.seed(0); n <- 25; x <- sort(rexp(n, 0.5))`, placing five knots by quantiles gives `k <- 5; g <- quantile(x, prob = seq(0, 1, length.out = k))`. The line plot is produced by lining up $(x_i,\ \frac{i-1}{n-1})$, i.e., `xp <- seq.int(0, n, length.out = n); plot(x, xp, type = "l"); points(x, xp, cex = 0.5)`. Consider slicing the vertical axis by `seq(0, 1, length.out = k)`, then the x-coordinates where these horizontal lines intersects the line plot are the quantiles (marked by bold solid dots on the x-axis). The slicing cuts the line into $k-1$ segments, and the graph above has boxed each piece by a rectangular. The number of data in each box is just the number of data between two adjacent knots. For this example there are 6, 6, 6, 7 data on the four segments (from left to right).

underfit the data; if k is too big, the spline can overfit the data. Ultimately when there are as many knots as data, we have an interpolation spline. The smoothing spline introduced in §1.1 does not end up with interpolation because of the penalization, which suppresses the effective degree of freedom of each spline coefficient to some values much smaller than one. A smoothing spline eliminates any knots related issues, but at the price of computational complexity. A low-rank penalized regression spline is less computationally expensive. Its basic idea is to practically choose a k, much smaller than the number of data, but big enough for the penalization to take effect.

To understand what this means, consider choosing k for our tensor product model 2.3. To facilitate the discussion here, I will rewrite it as

$$y_i = f_0(z_i;\ k_0) + f_1(z_i,\ x_i;\ k_{1,1},\ k_{1,2}) + \epsilon_i,$$

where k_0 is the number of knots for f_0, $k_{1,1}$ is the number of knots for z margin of f_1 and $k_{1,2}$ is the number of knots for x margin of f_1. Choosing those k values can be an iterative process. Here is an example demonstration.

1. Make an initial attempt with

   $$y_i = f_0(z_i;\ 4) + f_1(z_i,\ x_i;\ 4,\ 3) + \epsilon_i,$$

   then extract the partial residual w.r.t. $f_0(z_i,\ 4)$ and fit

   $$\text{partial residuals} = f_0(z_i,\ k_0) + \epsilon_i$$

   for a set of increasingly big k_0. For each trial record the number of coefficients (*ncoef* for short) in f_0 and the resulting effective degree of freedom (*edf* for short) of $\hat{f}_0$. The panel (a) of

Figure 2.7 sketches the *edf* against *ncoef*, with the corresponding k_0 labelled above the points. When penalization has no effect, no coefficients are suppressed so we should have *edf* $\approx$ *ncoef*. However, as k_0 grows we observe that *edf* plateaus, because the penalty starts to play its role. From the graph it seems that $k_0 = 16$ is an adaquate choice.

2. Update the model with $k_0 = 16$, i.e., fit

$$y_i = f_0(z_i;\ 16) + f_1(z_i,\ x_i;\ 4,\ 3) + \epsilon_i,$$

then extract the partial residual w.r.t. $f_1(z_i,\ x_i;\ 4,\ 3)$ and fit

$$\text{partial residuals} = f_1(z_i,\ x_i;\ k_{1z},\ 3) + \epsilon_i$$

for a set of increasing $k_{1,1}$. The panel (b) of Figure 2.7 sketches *edf* of $\hat{f}_1$ against its *ncoef*. It appears that $k_{1,1} = 12$ is sufficiently big.

3. Update the model with $k_{1,1} = 12$ i.e., fit

$$y_i = f_0(z_i;\ 16) + f_1(z_i,\ x_i;\ 12,\ 3) + \epsilon_i,$$

then extract the partial residual w.r.t. $f_1(z_i,\ x_i;\ 12,\ 3)$ and fit

$$\text{partial residuals} = f_1(z_i,\ x_i;\ 12,\ k_{1,2}) + \epsilon_i$$

for a set of increasing $k_{1,2}$. The panel (c) of Figure 2.7 sketches *edf* of $\hat{f}_1$ against its *ncoef*. It appears that $k_{1,2} = 10$ is big enough.

So in the end we have a model

$$y_i = f_0(z_i;\ 16) + f_1(z_i,\ x_i;\ 12,\ 10) + \epsilon_i.$$

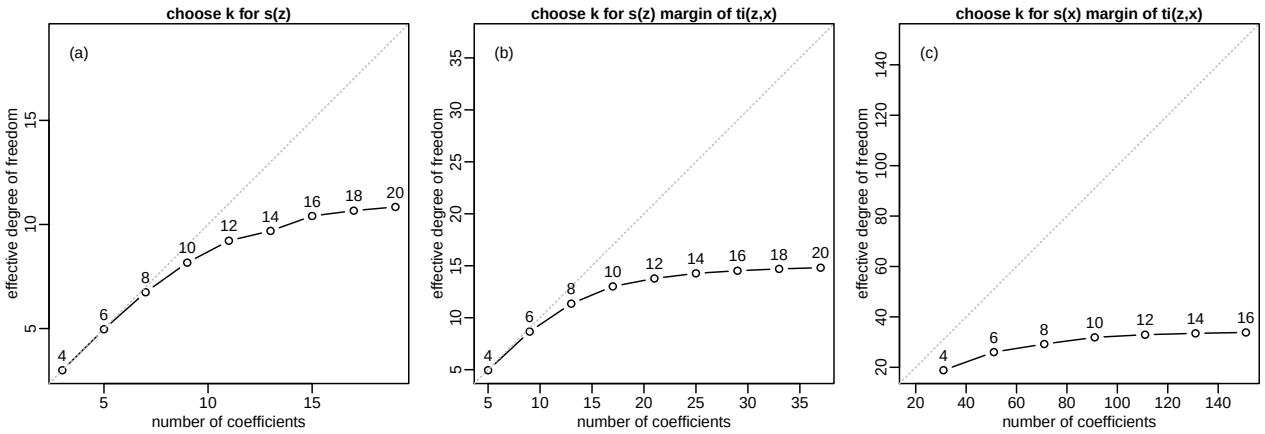


Figure 2.7: Choose the number of knots for model 2.3. These graphs illustrate the relationship between effective degree of freedom (*edf*, y-axis) and number of parameters (*ncoef*, x-axis), with the input k values labelled above the points. Due to penalization, The *edf* will not increase linearly with *ncoef*; it starts to plateau at some k value. This underlies the principle for choosing k. In practice an additive model has many spline terms so the selection process is a term-wise, iterative process. See main text for more explanations.

2.2 Time series example with AR(1) autocorrelation

This section adapts the example in §2.1 to a simple time series case study, replacing the i.i.d. error assumption with an AR(1) process with autocorrelation $\omega = 0.75$.

$$y_i = h(x_i) + e_i, \quad e_i = \omega \cdot e_{i-1} + \epsilon_i, \quad \epsilon_i \sim N(0,\ (1-\omega^2)\phi).$$

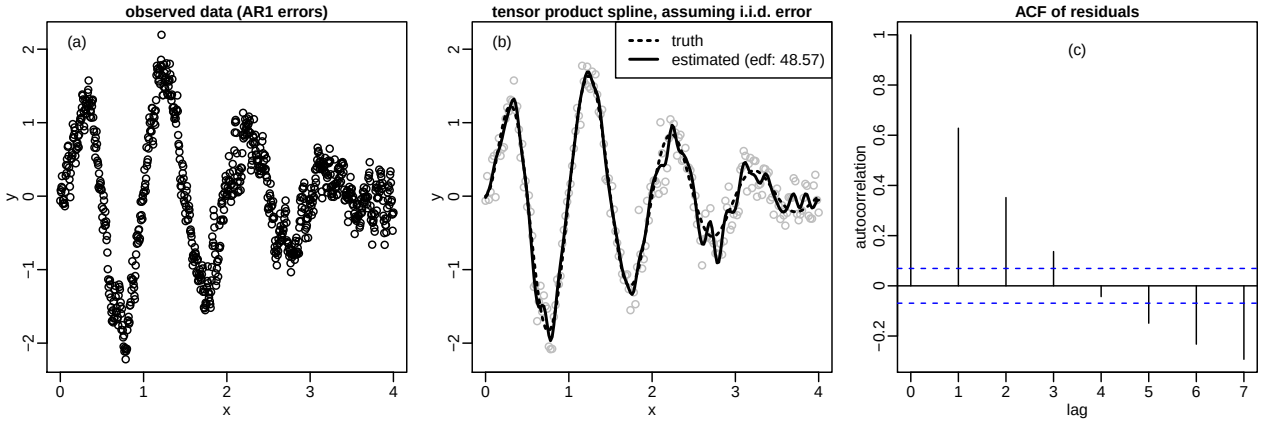


Figure 2.8: Example data used for §2.2. The `R` code to reproduce them is `set.seed(0); omega <- 0.75; x <- seq(0, 4, 0.005); A <- 2 * pi * x * exp(-2 * pi * x / 5); fx <- A * sin(2 * pi * x); y <- fx + base::c(arima.sim(list(ar = omega), n, sd = sqrt(1 - omega * omega) * 0.3 * sd(fx))); dat <- data.frame(x = x, z = x - floor(x), fx = fx, y = y)`. Panel (a) is created by `with(dat, plot(x, y))`. The tensor product spline (see model 2.3) in panel (b) is obtained by `fit <- gam(y ~ s(z, bs = "cc", k = 16) + ti(x, z, bs = c("cr", "cc"), k = c(12, 10)), data = dat, knots = list(z = c(0, 1)), method = "REML")`, but it ends up overfitting the data. The ACF in panel (c) is generated by `acf(fit$resi, lag.max = 7)`.

$\texttt{var}(\epsilon_i) = (1-\omega^2)\phi$ is specially chosen so that $\texttt{var}(e_i)$ is still ϕ, i.e., the noise-to-signal ratio is as same as the previous example. Panel (a) of Figure 2.8 sketches a random set of observed data, and see Figure caption for `R` code for reproducing it.

Autocorrelated error is known to cause overfitting in smoothing models, if errors are wrongly assumed to be i.i.d.. For example, see Krivobokova and Kauermann (2007); Opsomer et al. (2001); Smith et al. (1998); Wang (1998). Panel (b) of Figure 2.8 verifies this, where the tensor product spline model 2.3 assuming i.i.d. errors ends up overfitting the data. Panel (c) graphs the (sample autocorrelation function) ACF of model residuals. The residuals are still highly correlated even at presence of overfitting.

2.2.1 How to do exploratory analysis for data with autocorrelation

The existence of autocorrelation can complicate or even mislead model development. For example, let us consider choosing k for all splines in the tensor product model. In §2.1.3 it has been demonstrated that $k_1 = 16$, $k_{1,1} = 12$ and $k_{1,2} = 10$ are readily sufficient to approximate the truth. However, when observations are autocorrelated, this "fact" can be obscured. Figure 2.9 illustrates what will happen if we replicate exactly the same selection process on k as described in §2.1.3. Panel (a) shows that $k_1 = 16$ is far from being adequate; we probably need to choose $k_1 = 60$. Then in panel (b), the *edf* almost grows linearly and does not seem to plateau.

A useful trick for exploratory anaysis on data with autocorrelation is thinning. For example, let us examine every 4^{th} data indexed $i = 1,\ 5,\ 9, \cdots,\ 801$. Figure 2.10 shows that k can now be properly chosen. Figure 2.11 further shows that fitting the tensor product model 2.3 on the thinned data yields less misleading estimation.

2.2.2 AR(1) error with autocorrelation coefficient ω

When the model error e_i is an AR(1) process

$$e_i = \omega \cdot e_{i-1} + \epsilon_i, \quad \epsilon_i \sim N(0,\ (1-\omega^2)\phi),\ \ i.i.d.$$

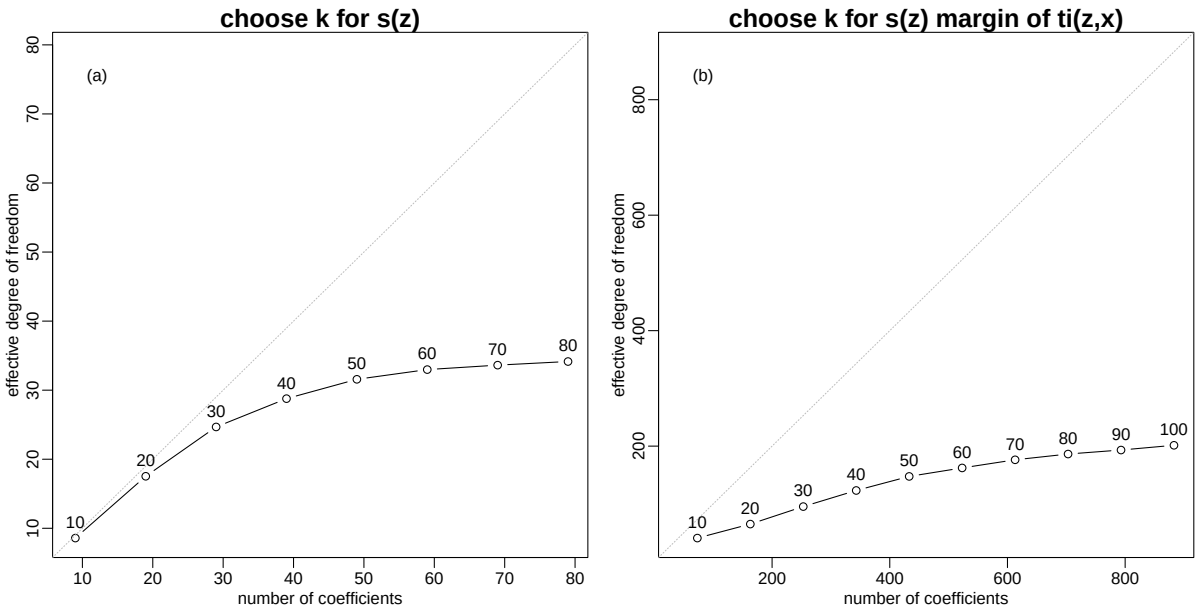


Figure 2.9: Choosing k can be misleading when working on data with autocorrelation. The degree of freedom (edf) may never plateau (see panel (b)).

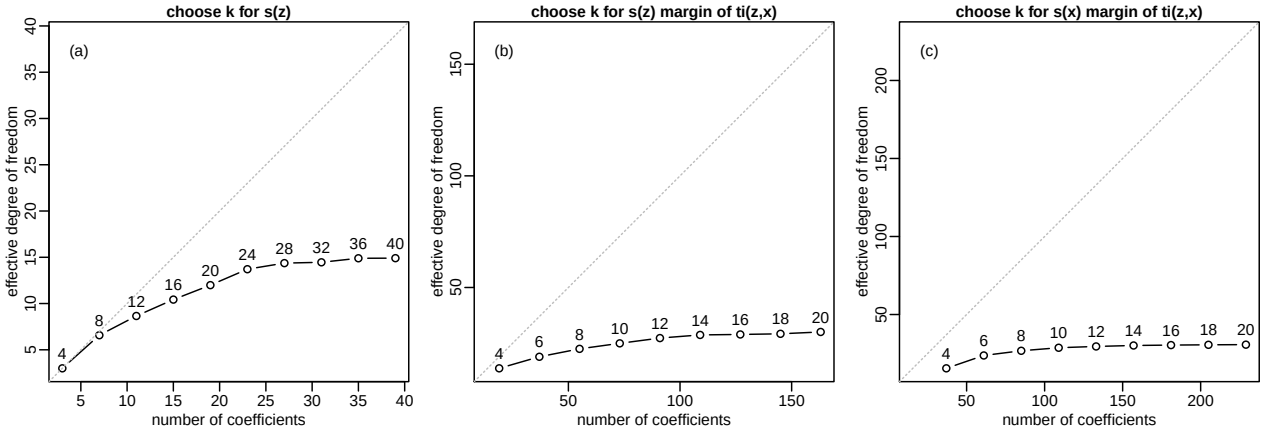


Figure 2.10: k can be properly chosen using thinned data indexed by $i = 1,\ 5,\ 9, \cdots,\ 801$.

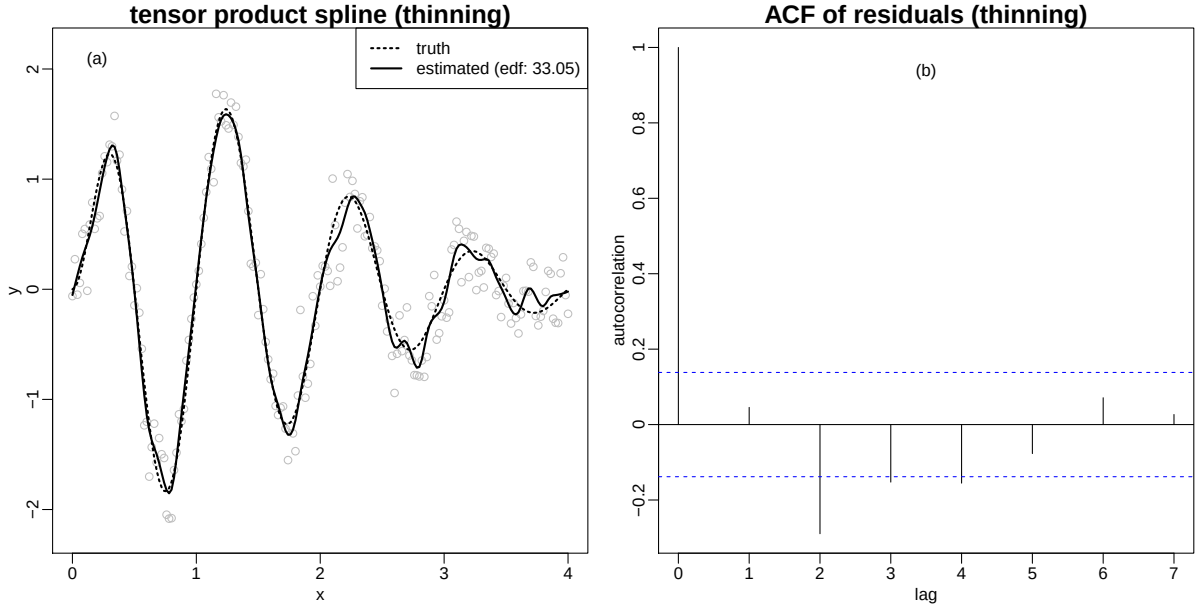


Figure 2.11: Fitting the tensor product model 2.3 for thinned data points indexed by $i = 1,\ 5,\ 9, \cdots,\ 801$ reduces overfitting. Residuals also do not exhibit autocorrelation. (Compare with Figure 2.8)

the resulting smoothing or additive model (1.3) has a symmetric tri-diagonal weight matrix (Rue and Held, 2005, §1.1.1):

$$\boldsymbol{W} = \begin{pmatrix} 1 & -\omega & & & \\ -\omega & 1+\omega^2 & -\omega & & \\ & \ddots & \ddots & \ddots & \\ & & -\omega & 1+\omega^2 & -\omega \\ & & & -\omega & 1 \end{pmatrix} \frac{1}{1-\omega^2}, \tag{2.1}$$

with an upper triangular Cholesky factor

$$\boldsymbol{W}^{\frac{1}{2}} = \begin{pmatrix} 1 & -\omega & & & \\ & 1 & -\omega & & \\ & & \ddots & \ddots & \\ & & & 1 & -\omega \\ & & & & \sqrt{1-\omega^2} \end{pmatrix} \frac{1}{\sqrt{1-\omega^2}}.$$

If ω is known a priori, $\boldsymbol{W}^{\frac{1}{2}}$ involves no unknown quantity hence the resulting smoothing or additive model (1.3) can be estimated using methods introduced in §4.2 and §4.3.

The matrix $\boldsymbol{W}^{\frac{1}{2}}$ is practically useful for data transformation.

- If $\boldsymbol{y}$ is a vector of AR(1) samples, i.e., $\texttt{var}(\boldsymbol{y}) = \boldsymbol{W}^{-1}\phi$, then $\boldsymbol{z} = \boldsymbol{W}^{\frac{1}{2}}\boldsymbol{y}$ is a vector of i.i.d. samples since $\texttt{var}(\boldsymbol{z}) = \boldsymbol{I}\phi$.
- If $\boldsymbol{z}$ is a vector of i.i.d. samples with $\texttt{var}(\boldsymbol{z}) = \boldsymbol{I}\phi$, then $\boldsymbol{y} = \boldsymbol{W}^{-\frac{1}{2}}\boldsymbol{z}$ is a vector of AR(1) samples since $\texttt{var}(\boldsymbol{y}) = \boldsymbol{W}^{-1}\phi$.

The first relation is often used for residual standardization, while the second is often used for simulating AR(1) samples.

First of all, at the beginning of §4.3 it is mentioned that the $\log|\boldsymbol{W}|$ in the full REML score (1.9) is dropped, so the REML score reported at convergence of the Newton-Raphson iteration (see Figure 4.9 if you need a revision) needs be "corrected" by adding back $\log|\boldsymbol{W}|$ which contains the information of ω. This is actually very straightforward. From the previous section, §2.2.2, we can derive $|\boldsymbol{W}^{\frac{1}{2}}| = (1-\omega^2)^{-\frac{n-1}{2}}$, so $\log|\boldsymbol{W}| = 2\log|\boldsymbol{W}^{\frac{1}{2}}| = -(n-1)\log(1-\omega^2)$. In the following I will denote the corrected REML score by $\tilde{\mathcal{V}}_r(\omega)$.

2.2.3 Golden-section search for point estimation of ω

Generally ω is unknown and needs be estimated as part of model estimation. An obvious solution is to perform a grid search on ω. For example, we fit the smoothing or additive model (1.3) for each of $\omega = 0.01,\ 0.02,\ 0.03,\ \cdots,\ 0.99$, then choose the one that minimizes the REML score. However, the grid search is very inefficient. For example, if we search from 0.01 to 0.99, the interval that contains the minimizer is [0.01, 0.99], [0.02, 0.99], [0.03, 0.99], ... which shrinks toward one direction slowly after each trial. A better idea is to use something like a bisection that is able to squeeze the interval from both directions. Suppose we evaluate $\tilde{\mathcal{V}}_r(\omega)$ at 0.25 and 0.75, then we can exclude one of the three intervals [0.01, 0.25], [0.25, 0.75] and [0.75, 0.99] from further searching, based on the relationship between $\tilde{\mathcal{V}}_r(0.25)$ and $\tilde{\mathcal{V}}_r(0.75)$.

1. If $\tilde{\mathcal{V}}_r(0.25) > \tilde{\mathcal{V}}_r(0.75)$, the minimizer can not be within [0.01, 0.25], otherwise $\tilde{\mathcal{V}}_r(\omega)$ should be non-decreasing on [0.25, 0.99], violating $\tilde{\mathcal{V}}_r(0.25) > \tilde{\mathcal{V}}_r(0.75)$. Therefore further searching can be restricted on [0.25, 0.99].

$r = (\sqrt{5} - 1)/2$
initialization
$c = b - r(b - a)$
$d = a + r(b - a)$
$f_c = f(c)$
$f_d = f(d)$
start iteration
while $(|c - d| > \epsilon)$
    if $(f_c > f_d)$
        shrinkage [a, b] to [c, b]
        $a = c$
        $c = d$
        $f_c = f_d$
        $d = a + r(b - a)$
        $f_d = f(d)$
    else
        shrinkage [a, b] to [a, d]
        $b = d$
        $d = c$
        $f_d = f_c$
        $c = b - r(b - a)$
        $f_c = f(c)$
return the mid point of [a, b] as the minimizer
$(a + b)/2$

Figure 2.12: Golden-section search algorithm for minimizing a univariate objective function $f(x)$ on interval $[a,\ b]$. The minimizer is always inside $[a,\ b]$ on each update of this searching interval. In the end the mid point of this interval is returned as the minimizer. ϵ is a parameter that controls the precision of result.

2. If $\tilde{\mathcal{V}}_r(0.25) < \tilde{\mathcal{V}}_r(0.75)$, the minimizer can not be within [0.75, 0.99], otherwise $\tilde{\mathcal{V}}_r(\omega)$ should be non-increasing on [0.01, 0.75], violating $\tilde{\mathcal{V}}_r(0.25) < \tilde{\mathcal{V}}_r(0.75)$. Therefore further searching can be restricted on [0.01, 0.75].

This interval shrinkage goes on iteratively. In general, we take two different points c and d in interval $[a,\ b]$, then squeeze the new searching interval to either $[c,\ b]$ or $[a,\ d]$ based on the relationship between $\tilde{\mathcal{V}}_r(c)$ and $\tilde{\mathcal{V}}_r(d)$. There are many ways to choose c and d (for example, we can set them randomly), but there exists an optimal choice (in terms of convergence speed) for c and d, given by $c = b - r(b - a)$ and $d = a + r(b - a)$, where $r = (\sqrt{5} - 1)/2$. This is known as *golden-section search* since $1/r$ is the golden ratio. Figure 2.12 is a snippet of this algorithm for finding the minimizer of a univariate function f on $[a, b]$.

We can now apply the golden-section search to integrate estimation of ω in our estimation of the tensor product model 2.3 for the simulated dataset in this section (§2.2). Figure 2.13 illustrates this, and the method proves successful. The point estimate $\hat{\omega} \approx 0.77$ is close to true value 0.75, and the fitted spline does not exhibit evident overfitting. Figure 2.14 further give the ACF and PACF of raw model residuals of the fitted model, as well as the ACF or standardized residuals. It is clear that the fitted model is adequate since there is no unmodelled autocorrelation in standardized residuals.

2.3 Visualization of a tensor product spline

Readers might be interested in knowing what the tensor product spline $f_1(z,\ x)$ in the fitted model 2.3 looks like. Figure 2.15 visualizes this surface function with a perspective plot, where surface facets

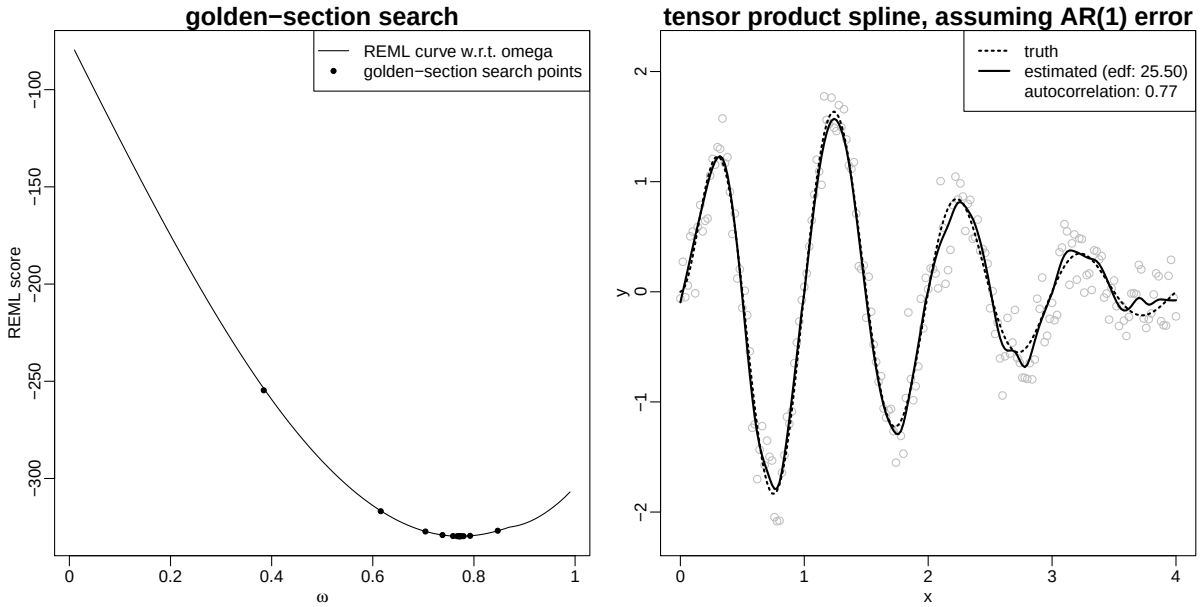


Figure 2.13: Implementation of golden-section search to integrate estimation of ω in estimation of the tensor product model 2.3 for the dataset in section (§2.2). The left panel sketches the curve of $\tilde{\mathcal{V}}_r(\omega)$, with the search path of golden-section marked as solid dots on the curve. Convergence occurs after 14 iterations, giving a point estimate of $\hat{\omega} = 0.7711866$ which is very close to the true value 0.75. The right panel illustrates the fitted spline function. The overfitting problem previously seen in Figure 2.8 is much alleviated.

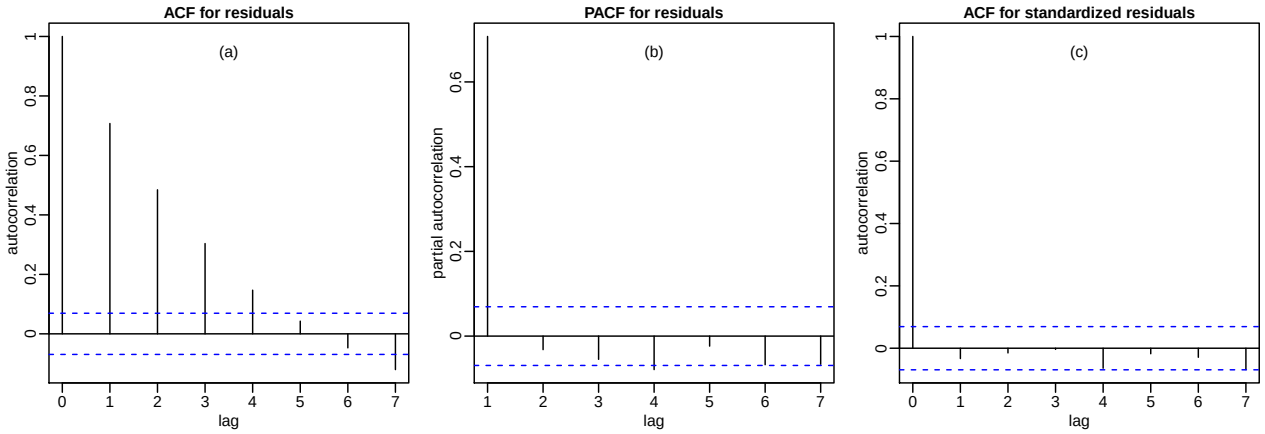


Figure 2.14: Residual inspection of fitted model 2.3 (with golden-section search integrated) for the dataset in section (§2.2). Panel (a) and (b) are respectively ACF and PACF of raw residuals, which convince the AR(1) correlation. Panel (c) is ACF of the standardized residuals (see §2.2.2 for how standardization is performed). The fitted model is adequate as there is no unmodelled autocorrelation.

are coloured according to function values. In the rest of this thesis, this will be the visualization method for any 2D surface smooth functions.

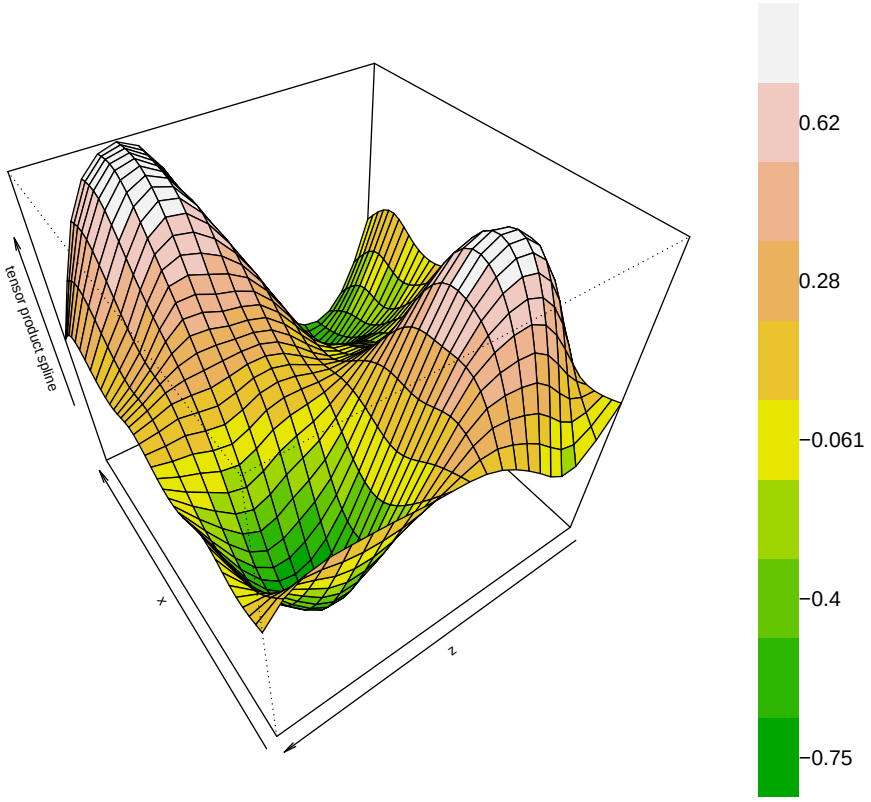


Figure 2.15: Visualization of the tensor product spline $f_1(z,\ x)$ in the fitted model 2.3 with a perspective plot, where surface facets are coloured according to function values (see the colour bar on the right for reference).

2.4 Model checking

In this section I will perform some model checking on the fitted time series model in §2.2.

First of all, let us check whether there is unmodelled trend w.r.t. a variable, say x. This is commonly visualized by plotting residuals against x, as is done in panel (a) of Figure 2.16. The scatter plot method is generally inconvenient if there are a great many of data points, as producing the plot takes much time and the resulting cloud of points obscures the interpretation. In this case, it is a good idea to bin the data points and produce a plot for summarized data. For example, panel (b) of Figure 2.16 cuts x values into 10 bins; residuals are also grouped by bins. Then 25% and 75% sample quantiles for residuals are computed on each bin. A gray shaded box whose height extends to those quantiles is displayed. Sample median is marked by a bold black dot in a box. The whiskers adds 1.5 times the interquartile (IQR) to the 75% sample quantile (the top of a box) and subtract 1.5 times the IQR from the 25% sample quantile (the bottom of a box). In the graph, samples within whiskers are not displayed at all; those outside the whiskers are seen as outliers and shown by gray dots. This plotting method substantially reduces the number of points to display, yet well illustrating the variability of data. A smoothing spline is fitted to the original data values (not summarized ones) and overlaid on the plot, for evidence of any trend. From the graph it appears that there is very complicated trend, however, this is because that the smoothing spline is smoothing the AR(1) residuals. Panel (c) produces the same plot using standardized residuals, and it is now clear that there is no unmodelled trend conditional on the AR(1) correlation.

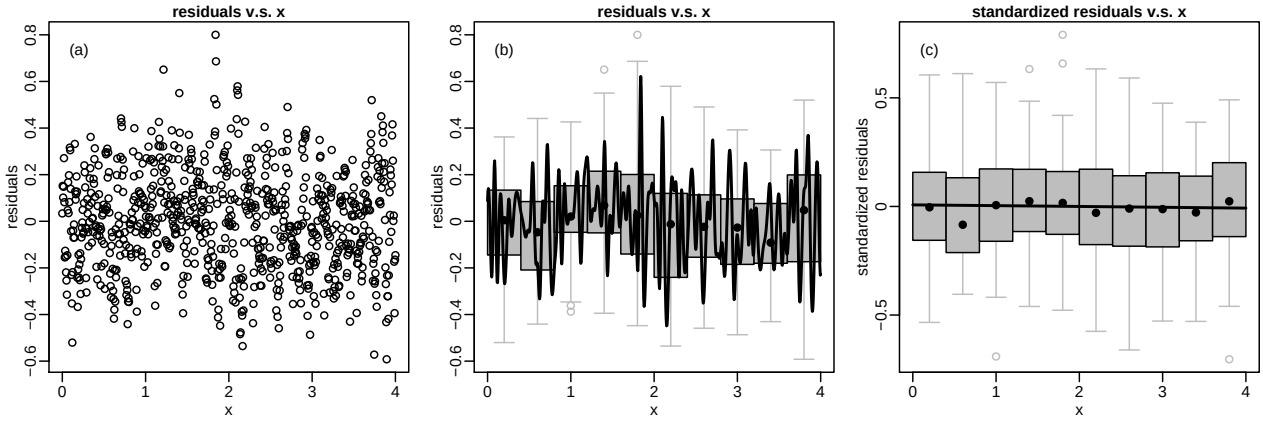


Figure 2.16: Inspecting residuals of fitted model 2.3 (with golden-section search integrated) for example data in §2.2 for any unmodelled trend w.r.t. variable x. Panel (a) is a straightforward scatter plot, while the boxplot-alike visualization in panel (b) and panel (c) are more useful for large dataset. See main text for detailed explanation on those plots. The over complicated trend spotted in panel (b) is due to the fact that the smoothing spline is fitted the AR(1) process in residuals. From panel (c), it seems that there is no unmodelled trend conditional on the AR(1) correlation.

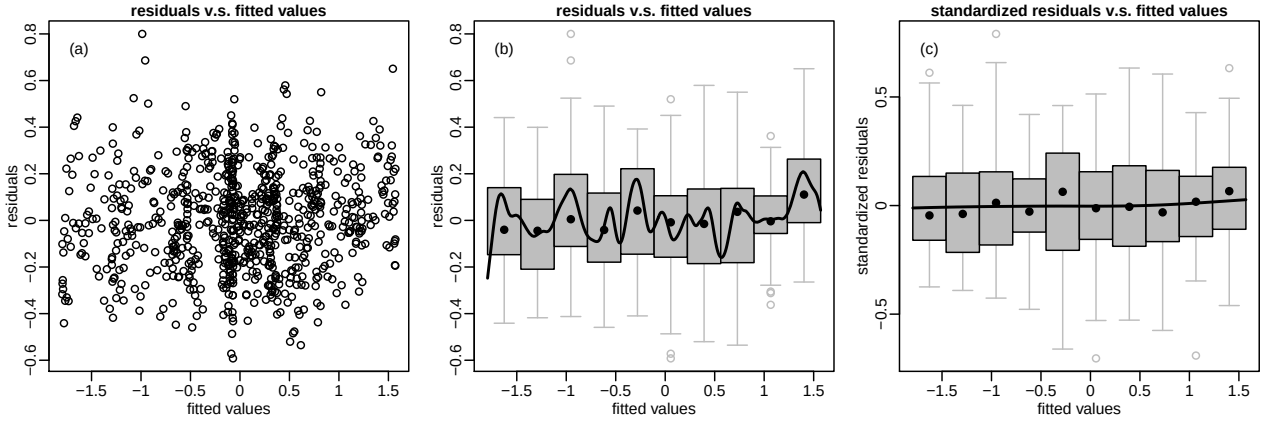


Figure 2.17: Inspecting residuals of fitted model 2.3 (with golden-section search integrated) for example data in §2.2 for constant error variance. Panel (a) is a straightforward scatter plot, while the boxplot-alike visualization in panel (b) and panel (c) are more useful for large dataset. See main text for detailed explanation on those plots. The spurious dependence of variance of mean spotted in panel (b) is due to the fact that the smoothing spline is fitted the AR(1) process in residuals. From panel (c), it seems that there is no violation of mean-variance independence conditional on the AR(1) correlation.

Secondly, let us check that residuals have constant variance (i.e., variance is independent of mean). This is often visualized by sketching residuals against fitted values. Panel (a) of Figure 2.17 is a straightforward scatter plot, while panel (b) and panel (c) are respectively boxplot-alike visualization using residuals and standardized residuals. Again, there does not seem to be any dependence of variance on mean.

2.5 Summary

In this Chapter we get some hands-on experience on smoothing and additive models via penalized regression splines.

In §2.1.1 I have demonstrated how to model time-varying cyclic pattern in data using a tensor product spline between a cubic cyclic spline and cubic regression spline. This model representation idea can be very useful in modelling time series data with trend and seasonality. While it might first appear that a simpler model representation via a single cubic regression spline is more straightforward, it is

in fact more sensitive to sampling locations of data hence less robust to unevenly spaced sampling, as is illustrated in §2.1.2.

The process of finding appropriate number of knots, k, for all splines in a model as is demonstrated in §2.1.3 is tedious, and may in fact be time consuming. This contributes to the non-negligible hidden costs for nonparametric modelling in practice.

The data thinning trick introduced in §2.2.1 is very useful for exploratory analysis of data with autocorrelation. In particular, when choosing k for splines in a model, we are less likely to overestimate the model complexity using thinned data. Correctly assessing model complexity is very important for model development. If an overcomplicated initial model is built at the beginning, the computational burden will be carried forward when new model terms are added and tested.

The Golden-section search method and implementation presented in §2.2.3 integrates the estimation of splines and the estimation of autocorrelation coefficient ω. Let $\mathcal{V}_r(\boldsymbol{\theta})$ be the REML score used in §4.3, the full REML score (in line with (1.9)) is then $\tilde{\mathcal{V}}_r(\boldsymbol{\theta},\ \omega) = \mathcal{V}_r(\boldsymbol{\theta}) - \log|\boldsymbol{W}(\omega)|$. The implementation minimizes $\tilde{\mathcal{V}}_r(\boldsymbol{\theta},\ \omega)$ w.r.t. $(\boldsymbol{\theta},\ \omega)$ by a nested iteration. The outer iteration is a one-dimensional minimization w.r.t. ω. Then for any trial value $\omega^{[k]}$, the inner iteration is a q-dimensional minimization w.r.t. $\boldsymbol{\theta}$ (assuming that the vector $\boldsymbol{\theta}$ has q elements):

$$\boldsymbol{\theta}^{[\boldsymbol{k}]}(\omega^{[k]}) = \underset{\boldsymbol{\theta}}{\arg\min}\ \left\{\mathcal{V}_r(\boldsymbol{\theta}) - \log|\boldsymbol{W}(\omega^{[k]})|\right\} = \underset{\boldsymbol{\theta}}{\arg\min}\ \mathcal{V}_r(\boldsymbol{\theta}).$$

Given this $\boldsymbol{\theta}^{[\boldsymbol{k}]}(\omega^{[k]})$, the full REML score becomes a univariate objective function:

$$\tilde{\mathcal{V}}_r\big(\boldsymbol{\theta}^{[\boldsymbol{k}]}(\omega^{[k]}),\ \omega^{[k]}\big) = \mathcal{V}_r\big(\boldsymbol{\theta}^{[\boldsymbol{k}]}(\omega^{[k]})\big) - \log|\boldsymbol{W}(\omega^{[k]})|,$$

so that the outer iteration can proceed. The inner iteration is performed with Newton-Raphson algorithm, and the outer iteration is performed with golden-section search. In principle this is not the best idea for such joint estimation. We may derive derivatives of $\tilde{\mathcal{V}}_r(\boldsymbol{\theta},\ \omega)$ w.r.t. $(\boldsymbol{\theta},\ \omega)$ so that a single $(q+1)$-dimensional minimization can be performed by Newton-Raphson algorithm. Unfortunately, this joint estimation introduces dependency between ω and regression coefficients, i.e., we now have $\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda},\boldsymbol{\omega}}$ instead of just $\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}$, so that REML derivatives presented in §4.3 are no longer valid. While it is possible to derive all required derivatives, the work is non-trivial. By contrast, the nested iteration implementation is convenient. Firstly, nothing needs be changed to the REML estimation procedure introduced in §4.3; secondly, the outer iteration, the golden-section search, requires no derivative computation at all. The only drawback is that model estimation is very computationally expensive. If the golden-section search takes 14 steps for convergence, we need to perform model matrix reduction for 14 times. And if REML estimation for every trial $\omega^{[k]}$ takes 16 steps for convergence, derivatives of REML score need be computed $14 \times 16 = 224$ times in total! I will make further remarks on this issue in later Chapters.

In §2.3 and §2.4 some data visualization methods are described. The boxplot alike plot as in Figure 2.16 and Figure 2.17 is a very useful alternative to a large scatter plot with thousands or even millions of dots (i.e., a "cloud" of dots).

In the next Chapter, I will start investigating the Black Smoke dataset.

Chapter 3

Preliminary modelling of log Black Smoke (logBS)

In this Chapter, I will start building additive models for Black Smoke.

Firstly, the daily Black Smoke dataset with a number of variables will be introduced in §3.1.1. As a background, some introduction and analysis of the monitoring network is also provided.

Then in the subsequent sections, I will investigate Black Smoke from a few different angles.

- §3.2 will explore the Black Smoke observed at a single station. This mainly involves model development for time series data. In this regard, the time series example previously studied in §2.2 offers a good guidance. Measurements from station *Manchester 11* is used for a thorough case study, because this site had the longest monitoring history with the greatest number of data. If a reasonable model can be built for this station, there wouldn't be any difficulty adapting the same model to other stations.
- §3.3 will apply the time series model concluded from *Manchester 11* to a representative set of stations then pool the result. This also motivates a joint spatial-temporal modelling for all stations' data.
- §3.4 will introduce spatial-temporal modelling using annual mean Black Smoke. Daily Black Smoke data has high day-to-day variability and strong day-to-day correlation, however, once they are aggregated on a yearly basis, the variability of annual mean is substantially reduced and autocorrleation eliminated. The resulting dataset thus only has spatial autocorrelation, making spatial-temporal modelling less challenging.
- §3.5 will tackle a more difficult spatial-temporal modelling task, by looking at daily Black Smoke in year 1967. This year is chosen because it was when the Network had the most measurements: it had the most operating stations and most of them worked for 365 days. Models are fitted for logBS from each day of week to eliminate the impact of temporal autocorrelation in daily observations. It turns out much harder to adequately model space-time relationship without using odd-looking three-way interactions.

While these preliminary modelling exposes a few statistical questions on modelling Black Smoke, they all give way to the computational hurdles of GAM fitting. Modelling daily logBS requires high model complexity (i.e., great number of parameters) that the exisiting GAM computational engine is not able to deal with.

3.1 Introduction to daily logBS dataset and monitoring network

3.1.1 The daily logBS dataset

The raw daily Black Smoke dataset simply has three variables:

- the name of the monitoring station, like *Bath 6*, Manchester 11, etc;
- the date of the measurement, like the first day 1961-10-03 and the last day 2005-12-31;
- the Black Smoke measurements, which are non-negative integers.

Plenty of time in my PhD was spent in organizing and enriching the dataset. I was able to obtain the following key information / variables:

- A table providing spatial locations of all monitoring stations. Locations are given by Ordnance Survey grid reference as Northing and Easting. They are originally in metres but are converted to kilometres in my dataset.
- A polygon shapefile for boundaries of Bristish isles. I have processed and exported it as a simple two-column matrix providing the locations of polygon vertices in Northing and Easting.
- Elevation at all stations, extracted from Ordnance Survey's Terrain 50 model (Terrain-50, 2015).
- Daily temperature variables and monthly rainfall variable, extracted from UKCP09 gridded datasets (Perry et al., 2009; Perry and Hollis, 2006) produced by Met Office.

In my organized Black Smoke dataset, Black Smoke measurements (BS) are transformed to logarithmic scale by: $\text{logBS} = \log(\text{BS} + \delta)$. BS are non-negative integers which are highly skewed (with mean 47.2 and median 21). A log-transformation would make data more conformable to normality, thus additive models for logBS with Gaussian error assumption becomes appropriate. The choice of δ is essentially arbitrary, as long as it is not too big compared with BS. I have chosen $\delta = 1$ so that logBS has a minimum of 0, with mean 3.13 and median 3.09.

Date has been mapped to a number of time variables. There are two types of time variables. The first type is *cumulative* time variable, denoted by a single lowercase letter. These variables are:

- `d`, day since 1961-10-03. For example, 1961-10-03 is day 1, and 2005-12-31 is day 16131;
- `w`, week since 1961-10-02 (the nearest Monday to 1961-10-03), ranging from 1 to 2305.
- `m`, month since 1961-Jan, up to 2005-Dec. For example, 1961-10-03 is in month 10, 1962-10-03 is in month 22 and 2005-12-01 is in month 540.
- `y`, year, ranging from 1961 to 2005.

The second type is *nested* time variable, denoted by a fashion of $\texttt{u}_\texttt{v}$, which means `u` of `v`. These variables are:

- $\texttt{d}_\texttt{y}$, day of year, ranging from 1 to 366. For example, 1965-01-01 and 1975-01-01 are both day 1 of a year. 2000-12-31 is day 366 of the leap year 2000, while 2003-12-31 is day 365 of the common year 2003.

- $\mathtt{d_w}$, day of week, ranging from 1 to 7, for Monday to Sunday.
- $\mathtt{w_y}$, week of year, ranging from 0 to 53. This variable needs some explanation. I define that the first week of a year starts from the nearest Monday to the first day of that year. For example, 1989-01-01 is Sunday, so it is in week 0 of the year and week 1 starts from 1989-01-02. As another example, 2019-01-01 is Tuesday, so it is week 1 of the year and there is no week 0.

The dataset has the following variables associated with monitoring stations:

- $\mathtt{i}$, numeric station ID, from 1 to 2874.
- $\{\mathtt{e_i},\ \mathtt{n_i}\}$, spatial locations of station$\mathtt{i}$;
- $\mathtt{h_i}$, elevation of station $\mathtt{i}$;
- $\mathtt{E_i}$, environmental type at $\mathtt{i}$. According to Loader (2002), environmental type has five major levels: residential area with high-density housing (A), suburb residential area with medium-density housing (B), industrial area (C), commercial area or town centre (D) and rural community or open country (R).

For meteorological variables, the dataset has

- $\mathtt{T^0_{id}}$, daily minimum temperature (°C) for station $\mathtt{i}$ and day $\mathtt{d}$;
- $\mathtt{T^1_{id}}$, daily maximum temperature (°C) for station $\mathtt{i}$ and day $\mathtt{d}$;
- $\mathtt{T^*_{id}}$, diurnal temperature variation for station $\mathtt{i}$ and day $\mathtt{d}$, defined by $\mathtt{T^1_{id}} - \mathtt{T^0_{id}}$;
- $\mathtt{r_{im}}$, monthly total rainfall (milimetre) for station $\mathtt{i}$ and month $\mathtt{m}$.

In total, the dataset has 9489903 (about 10 million) data from a total of 2874 stations for 45 years from 1961-10-03 to 2005-12-31.

3.1.2 The Black Smoke monitoring network

Although the dataset has a big volume of data, the information over space and time was only sufficient for 20 years prior to 1981 or 1982. Figure 3.1 maps the Network on 1975, 1985, 1995 and 2005. To enhance visualization a gray shaded area within 10 km distance from all 2874 historical stations (I will hereafter call this the "local domain" of the Network) is also displayed. First of all, stations were not evenly spread out over the islands; secondly, even on the local domain the Network was getting thinner and thinner in later years.

Since Black Smoke declined over decades, It has been suspected that stations with lower Black Smoke observations were more likely to be shut down through years, making the Network sparser and sparser. Under the assumption that logBS declined with the same rate for all stations, then if this preferential sites closure existed, the sample spatial mean or other similar statistics summarized from operating stations alone should decline slower and slower, over even increase at some point. A common statistic for monitoring networks is *exceedance*, the number of days when Black Smoke exceeded a certain limit. Table 3.1 lists the Black Smoke limits imposed by EC Directive 80/779/EEC. This Directive was introduced to UK in 1980, exactly when the Network experienced a massive wave of sites' closure. The role of the Network had changed thereafter. The Network mannual states clearly that monitoring compliance with this Directive became the Network's main objective throughout the remainder of its

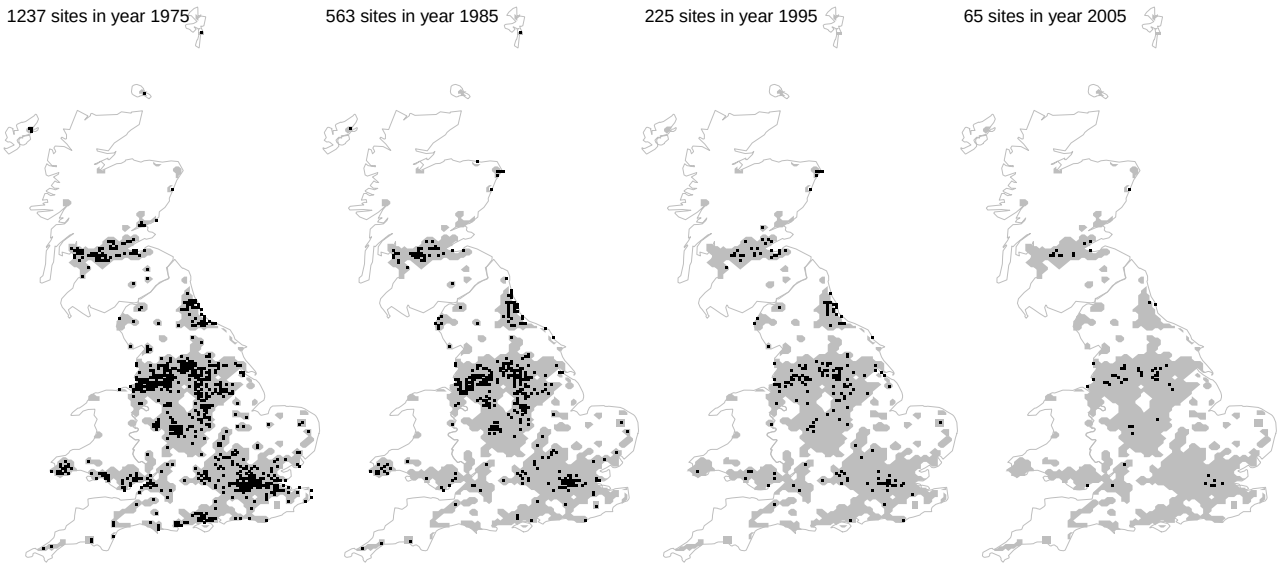


Figure 3.1: Black Smoke monitoring network (1961-2005). The maps outline the Network in 1975, 1985, 1995 and 2005 by landmarking operational sites in black dots. The gray shaded area covers regions within 10 km distance from all 2874 historical stations (I will call it a "local domain" of the Network).

Table 3.1: Directive 80/779/EEC Limits for Black Smoke (1980 $\sim$ 2004)

reference period	Limits (μgm^{-3})
Year (median of daily values)	68
Winter (median of daily values Oct. - Mar.)	111
Year (Peak: 98 Percentile of daily values)	213

existence. (In fact, the final closure of the Network was also related to this Directive. On January 1st, 2005, this Directive was repealed, and the Network's monitoring role was ceased at the end of that year.)

Figure 3.2 sketches the percentage of sites failing to comply the EC Directive limits over years. It is interesting to observe that by 1980 when EC Directive was introduced, all three percetages had declined to almost zero. Then in 1981 and 1982, the percentage associated with annual peak limits suddenly rose up again. Since this was the time when the Network was reduced in size, it is almost surely an evidence that 1) sites where BS had been consistently below the limits for many years were closed; 2) only sites where Black Smoke was not low enough were retained for a few more years for monitoring.

In summary, the information carried by BS observations is unbalanced over the years.

3.2 Time series models for daily logBS from "MANCHESTER 11"

3.2.1 Description of data

In this section let us build a time series model for logBS from station *Manchester 11*, which had the longest monitoring history for the entire 45 years (1961 $\sim$ 2005) with 15707 data. Figure 3.3 is an illustration of this dataset.

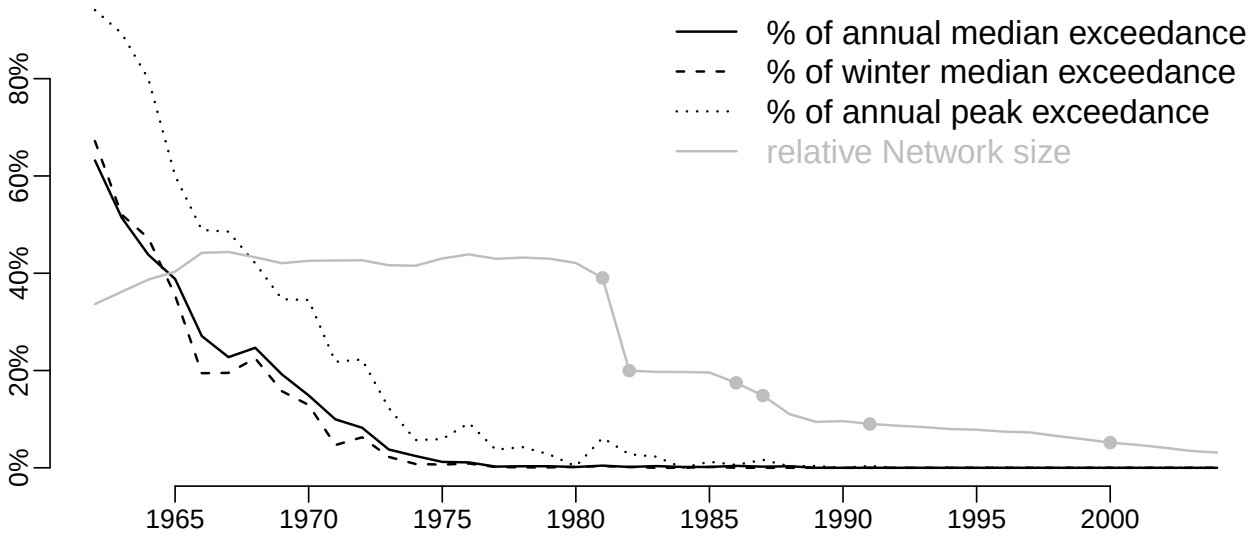


Figure 3.2: A profile of the Network. The gray solid line gives the relative size of the Network, defined as the ratio (in percentage) between the number of operational sites in a year and the total number of historical sites. The black solid, dashed and dotted lines respectively give the percentage of sites in a year failing to comply the EC Directive limits in annual median, winter median and annual peak (see Table 3.1).

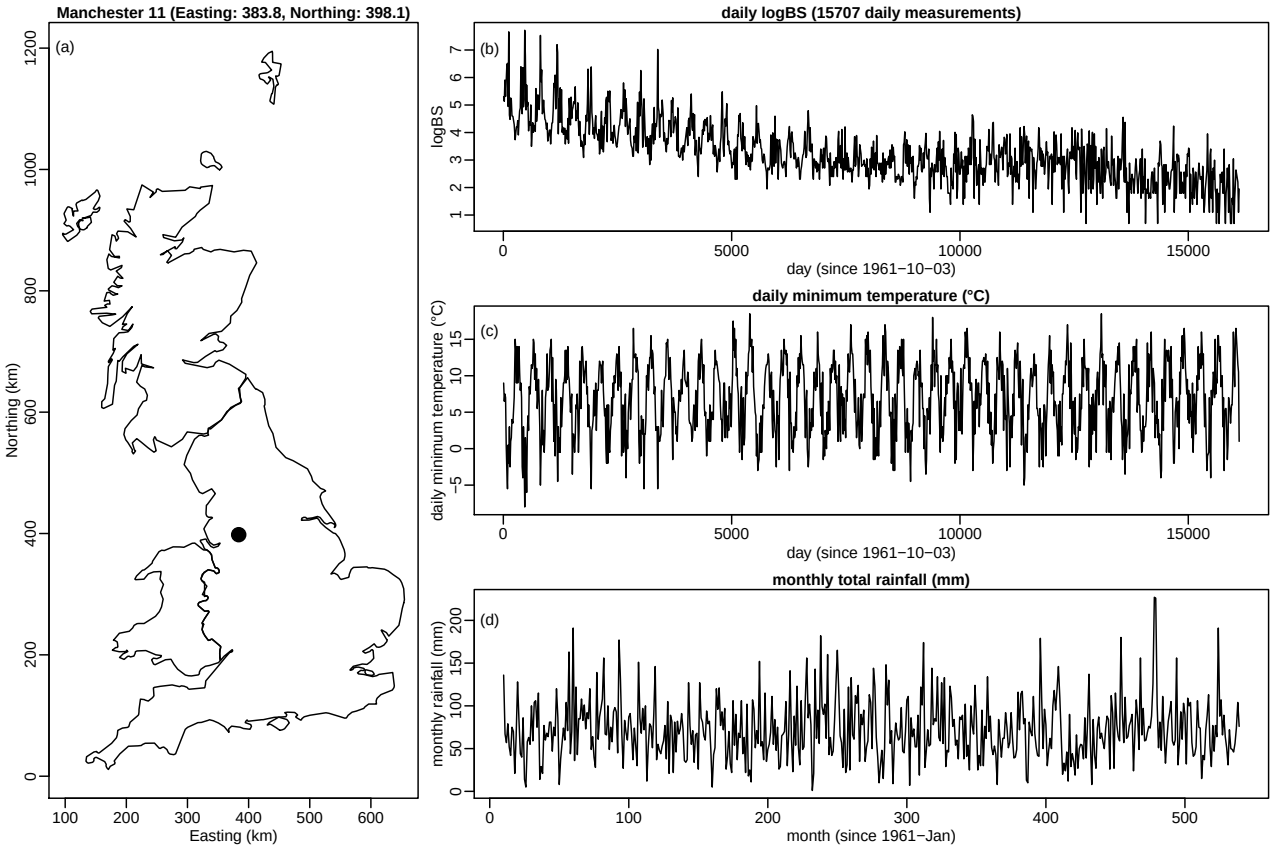


Figure 3.3: Data from monitoring station *Manchester 11*. Panel (a) is the location of this station on map; Panel (b) is the daily logBS time series; Panel (c) is the daily minimum temperature at this location; Panel (d) is the monthly total rainfall at this location.

3.2.2 Building a separate model for each day of week

It can be spotted from Panel (a) of Figure 3.3 that the logBS time series has an evident cyclic characteristic within year, i.e., seasonality. Inspired by the case study in §2.1, I propose the following model:

$$\texttt{logBS}_\texttt{d} = f_1(\texttt{d}_\texttt{y};\ k_1) + f_2(\texttt{d};\ k_2) + f_3(\texttt{d}_\texttt{y},\ \texttt{d};\ k_{3,1},\ k_{3,2}) + \epsilon_\texttt{d},$$

where

- $f_1(\texttt{d}_\texttt{y})$ is a cubic cyclic spline with k_1 number of knots, modelling the seasonality of logBS;
- $f_2(\texttt{d})$ is a natural cubic spline with k_2 number of knots, modelling the trend of logBS;
- $f_3(\texttt{d}_\texttt{y},\ \texttt{d})$ is a tensor product spline between a cubic cyclic spline margin for $\texttt{d}$ with $k_{3,1}$ knots and a natrual cubic spline margin for $\texttt{d}$ with $k_{3,2}$ knots, modelling how seasonality varies with time;
- $\epsilon_\texttt{d}$ is an i.i.d. Gaussian error on each day.

The i.i.d. error assumption is very implausible for daily data that usually have strong autocorrelation. So I will apply the data thinning trick explained in §2.2.1, by looking at data from for example, Monday only. That is, I will start by building a model for each day of week. Note that in this way, it is equivalent to use $\texttt{w}_\texttt{y}$ in place of $\texttt{d}_\texttt{y}$ and $\texttt{w}$ instead of $\texttt{d}$. So, I will write this model as

Model 3.1:

$$\texttt{logBS}_\texttt{w} = f_1(\texttt{w}_\texttt{y};\ k_1) + f_2(\texttt{w};\ k_2) + f_3(\texttt{w}_\texttt{y},\ \texttt{w};\ k_{3,1},\ k_{3,2}) + \epsilon_\texttt{w},$$

While model 3.1 is potentially a reasonable structure on thinned data, its required complexity, i.e., the number of knots needed for each spline is unknown yet. Following the "choose k" procedure explained in §2.1.3, I conducted the following selection process.

1. Fit model 3.1 (for each day of week) with $k_1 = 10$, $k_2 = 30$, $k_{3,1} = 10$ and $k_{3,2} = 30$. Extract the partial residual w.r.t. $\hat{f}_1(\texttt{w}_\texttt{y},\ 10)$, then fit model (for each day of week)

   $$\text{partial residuals} = f_1(\texttt{w}_\texttt{y},\ k_1) + \text{i.i.d. errors}$$

   for a set of increasingly big k_1. The panel (a) of Figure 3.4 implies that $k_1 = 15$ is sufficient for all days of week.

2. Fit model 3.1 (for each day of week) with $k_1 = 15$, $k_2 = 30$, $k_{3,1} = 10$ and $k_{3,2} = 30$. Extract the partial residual w.r.t. $\hat{f}_2(\texttt{w},\ 30)$ then fit model (for each day of week)

   $$\text{partial residuals} = f_2(\texttt{w},\ k_2) + \text{i.i.d. errors}$$

   for a set of increasingly big k_2. The panel (b) of Figure 3.4 implies that $k_2 = 40$ is sufficient for all days of week.

3. Fit model 3.1 (for each day of week) with $k_1 = 15$, $k_2 = 40$, $k_{3,1} = 10$ and $k_{3,2} = 30$. Extract the partial residual w.r.t. $\hat{f}_3(\texttt{w}_\texttt{y},\ \texttt{w};\ 10,\ 30)$ and fit model (for each day of week)

   $$\text{partial residuals} = f_3(\texttt{w}_\texttt{y},\ \texttt{w};\ k_{3,1},\ 30) + \text{i.i.d. errors}$$

   for a set of increasingly big $k_{3,1}$. The panel (c1) of Figure 3.4 implies that $k_{3,1} = 10$ is sufficient for all days of week.

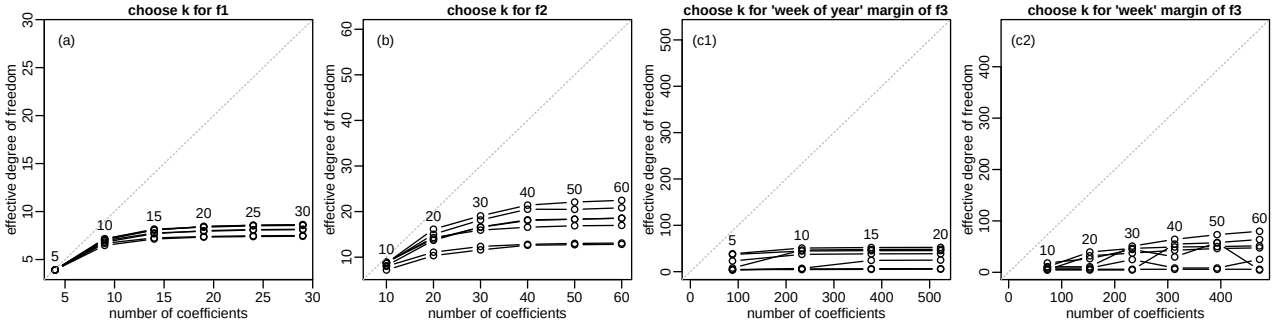


Figure 3.4: Choosing k_1, k_2, $k_{3\mathtt{d}}$ and $k_{3\mathtt{d}}$ for model 3.1. These graphs are similar to those in Figure 2.7, except that the results for all seven days of a week are displayed. This gives a good idea on a reasonable choice of k-values for all seven models.

Table 3.2: Number knots, coefficients and effective degree of freedom in fitted model 3.1, for each day of week.

			Mon	Tue	Wed	Thu	Fri	Sat	Sun
	No. of knots	No. of coef.	effective degree of freedom						
$\hat{f}_1$	$k_1 = 15$	13	7.0	6.1	7.1	6.7	6.3	6.7	7.0
$\hat{f}_2$	$k_2 = 40$	39	15.4	12.3	16.8	20.2	20.0	17.1	11.1
$\hat{f}_3$	$k_3 = (10, 40)$	$8 \times 39 = 312$	62.4	21.0	46.4	41.2	53.3	4.8	47.6

4. Update model 3.1 with $k_1 = 15$, $k_2 = 40$, $k_{3,1} = 10$ and $k_{3,2} = 30$. Extract the partial residual w.r.t. $\hat{f}_3(\mathtt{w_y},\ \mathtt{w};\ 10,\ 30)$ and fit model (for each day of week)

$$\text{partial residuals} = f_3(\mathtt{w_y},\ \mathtt{w};\ 10,\ k_{3,2}) + \text{i.i.d. errors}$$

for a set of increasingly big $k_{3,2}$. The panel (c2) of Figure 3.4 implies that $k_{3,2} = 40$ is sufficient for all days of week.

So an adequate model is

$$\mathtt{logBS_w} = f_1(\mathtt{w_y};\ 15) + f_2(\mathtt{w};\ 40) + f_3(\mathtt{w_y},\ \mathtt{w};\ 10,\ 40) + \epsilon_{\mathtt{w}},$$

and Table 3.2 summarizes the number of parameters and the resulting effective degree of freedom for each splines.

All seven fitted models turn out adequate. Figure 3.5 sketches the sample autocorrelation (ACF) of residuals for all models, and no significant autocorrelation is spotted.

The data thinning trick has indeed helped the selection of k. If using complete data, the complexity of model 3.1 will be severly overestimated. For example, k_1 must be at least 50 to be seen "sufficient" and k_2 needs be 100. That is, the spline function is fitting the autocorrelation in daily data.

3.2.3 A joint model for all days of week

Now I will build a joint model for all days of week. To start with, I introduce $\mathtt{d_w}$ as a factor of seven levels so that the intercept and all spline functions in model 3.1 can vary between levels. This model can be expressed as:

$$\mathtt{logBS_d} = \sum_{s=1}^{7} \delta_s^{\mathtt{d_w}} u_s + \sum_{s=1}^{7} \delta_s^{\mathtt{d_w}} f_{1,s}(\mathtt{w_y};\ 15) + \sum_{s=1}^{7} \delta_s^{\mathtt{d_w}} f_{2,s}(\mathtt{w};\ 40) + \sum_{s=1}^{7} \delta_s^{\mathtt{d_w}} f_{3,s}(\mathtt{w_y},\ \mathtt{w};\ 10,\ 40) + e_{\mathtt{d}},$$

where

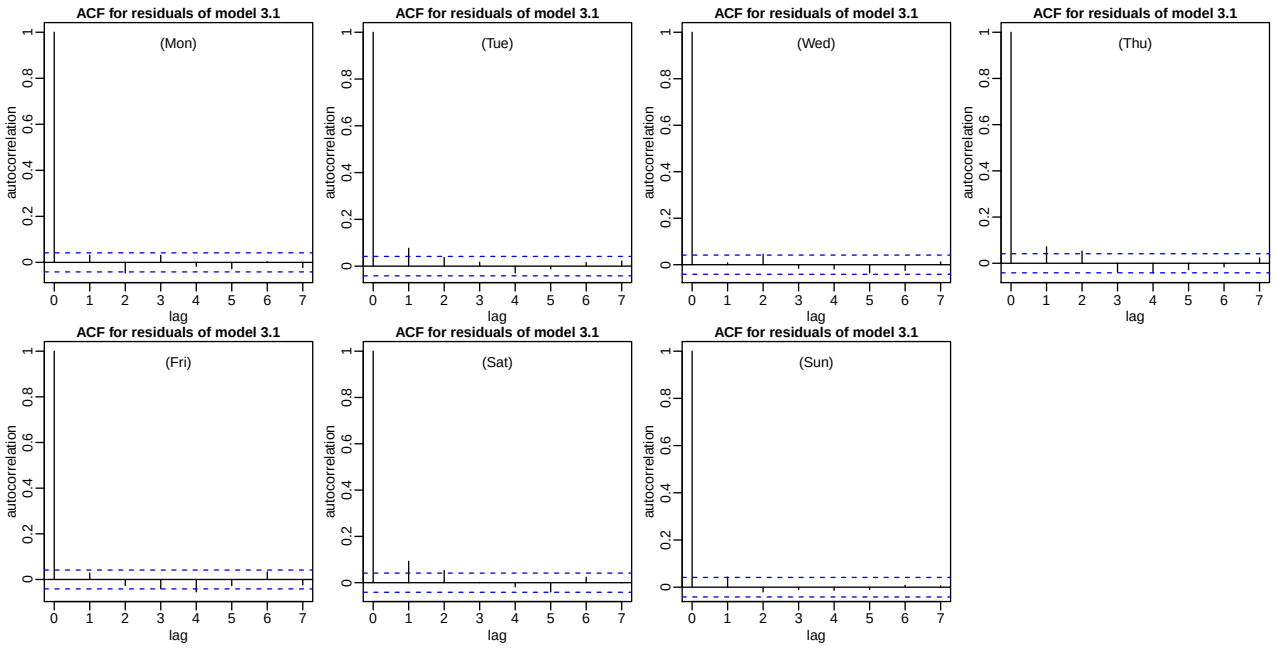


Figure 3.5: Sample autocorrelation (ACF) of residuals for model 3.1 fitted to each day of week. No significant autocorrelation is seen.

- u_s is the intercept for day s of week (so that there will be u_1, u_2, ..., u_7) and $\delta_s^{\mathtt{d_w}}$ is a dummy variable with value 1 when $s = \mathtt{d_w}$ and 0 otherwise;
- $f_{1,s}$ is a cubic cyclic spline of $\mathtt{w_y}$, modelling the seasonality of logBS from level s;
- $f_{2,s}$ is a natural cubic spline of $\mathtt{w}$, modelling the trend of logBS from level s;
- and $f_{3,s}$ is tensor product spline, modelling how seasonality of logBS from level s varies with time;
- $e_{\mathtt{d}}$ is a model error with some autocorrelation structure.

In the above specification, there is no contrasts applied to $\mathtt{d_w}$. An alternative specification is to set Monday as the reference level, so that spline functions for Monday are the reference smooth functions, and spline functions for other levels are the deviation from those reference smooth functions. This gives

Model 3.2a:

$$\begin{aligned}\mathtt{logBS_d} = u_1 + \sum_{s=2}^{7} \delta_s^{\mathtt{d_w}} u_s + f_1(\mathtt{w_y};\ 15) + \sum_{s=2}^{7} \delta_s^{\mathtt{d_w}} f_{1,s}(\mathtt{w_y};\ 15) + \\ f_2(\mathtt{w};\ 40) + \sum_{s=2}^{7} \delta_s^{\mathtt{d_w}} f_{2,s}(\mathtt{w};\ 40) + \\ f_3(\mathtt{w_y},\ \mathtt{w};\ 10,\ 40) + \sum_{s=2}^{7} \delta_s^{\mathtt{d_w}} f_{3,s}(\mathtt{w_y},\ \mathtt{w};\ 10,\ 40) + e_{\mathtt{d}},\end{aligned}$$

The advantange of this specification in GAM estimation, is that if there is little difference between day s and Monday, the "difference smooth" will be "penalized out", ending up with zero degree of freedom. This will be a sign that some components can be dropped from the model.

Fitting this additive model assuming that $e_{\mathtt{d}}$ is i.i.d. ends up with the effective degree of freedom for all functions as is summarized in Table 3.3. There are the following conclusion.

1. $f_{3,2}$, $f_{3,3}$, ..., $f_{3,7}$ are "penalized out", so they can be dropped from the model.

Table 3.3: Effective degree of freedom for all functions in fitted model 3.2a, assuming i.i.d. model error.

$\hat{f}_1$	$\hat{f}_{1,2}$	$\hat{f}_{1,3}$	$\hat{f}_{1,4}$	$\hat{f}_{1,5}$	$\hat{f}_{1,6}$	$\hat{f}_{1,7}$
11.58	0.00	0.10	0.75	1.58	0.98	2.23
$\hat{f}_2$	$\hat{f}_{2,2}$	$\hat{f}_{2,3}$	$\hat{f}_{2,4}$	$\hat{f}_{2,5}$	$\hat{f}_{2,6}$	$\hat{f}_{2,7}$
25.33	1.00	2.53	2.90	2.22	1.00	6.58
$\hat{f}_3$	$\hat{f}_{3,2}$	$\hat{f}_{3,3}$	$\hat{f}_{3,4}$	$\hat{f}_{3,5}$	$\hat{f}_{3,6}$	$\hat{f}_{3,7}$
235.79	0.00	0.00	0.00	0.00	0.00	0.00

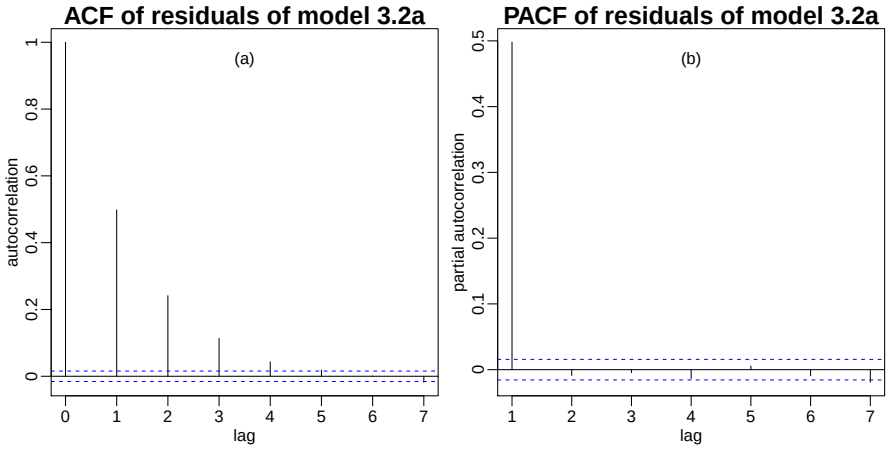


Figure 3.6: Strong residual autocorrelation is spotted in residuals of fitted model 3.2a. Panel (a) is the ACF and panel (b) is the PACF. The cutoff of partial autocorrelation at lag 1 implies that residuals are AR(1) correlated.

2. Not all of $f_{2,2}$, $f_{2,3}$, ..., $f_{2,7}$ are "penalized out", so they should all be retained.
3. Compared with Table 3.2, the effective degree of freedom for f_1, f_2 and f_3 have increased. This is a message that those spline functions are fitting autocorrelation in daily data.

In fact, the existence of strong autocorrelation can be verified from model residuals, as is shown in Panel (a) of Figure 3.6. The exponential decay of autocorrelation w.r.t. time lag probably implies that residuals are AR(1) correlated. Panel (b) confirms this with the cutoff of partial autocorrelation at lag 1.

These observations lead to the following model:

Model 3.2b:

$$\texttt{logBS}_\texttt{d} = u_1 + \sum_{s=2}^{7} \delta_s^{\texttt{d}_\texttt{w}} u_s + f_1(\texttt{w}_\texttt{y};\ 15) + \sum_{s=2}^{7} \delta_s^{\texttt{d}_\texttt{w}} f_{1,s}(\texttt{w}_\texttt{y};\ 15) + f_2(\texttt{w};\ 40) + \sum_{s=2}^{7} \delta_s^{\texttt{d}_\texttt{w}} f_{2,s}(\texttt{w};\ 40) + f_3(\texttt{w}_\texttt{y},\ \texttt{w};\ 10,\ 40) + e_\texttt{d},$$

where $f_{3,2}$ to $f_{3,7}$ in model 3.2a are dropped and $e_\texttt{d}$ is assumed an AR(1) stochastic process with some unknown autocorrelation coefficient ω. The golden-section search implementation introduced in §2.2.3 makes estimation of this additive model possible. It gives a point estimate $\hat{\omega} = 0.5447645$ (see 3.8 for the search path of golden-section), and the resulting degree of freedom for all spline functions in the model are reported in Table 3.4. Note that the values for f_1 and f_2 are now similar to those in Table 3.2. This is a message that autocorrelation has been properly modelled and the spline functions are not overfitting data. Figure 3.7 is a formal residual checking, where AR(1) assumption is validated.

Table 3.4: Effective degree of freedom for all functions in fitted model 3.2b, assuming AR(1) error with unknown autocorrelation coefficient.

$\hat{f}_1$	$\hat{f}_{1,2}$	$\hat{f}_{1,3}$	$\hat{f}_{1,4}$	$\hat{f}_{1,5}$	$\hat{f}_{1,6}$	$\hat{f}_{1,7}$
7.83	0.00	0.00	0.00	1.40	1.39	2.38
$\hat{f}_2$	$\hat{f}_{2,2}$	$\hat{f}_{2,3}$	$\hat{f}_{2,4}$	$\hat{f}_{2,5}$	$\hat{f}_{2,6}$	$\hat{f}_{2,7}$
19.8	2.25	3.03	3.49	2.75	1.00	8.79
$\hat{f}_3$						
81.65						

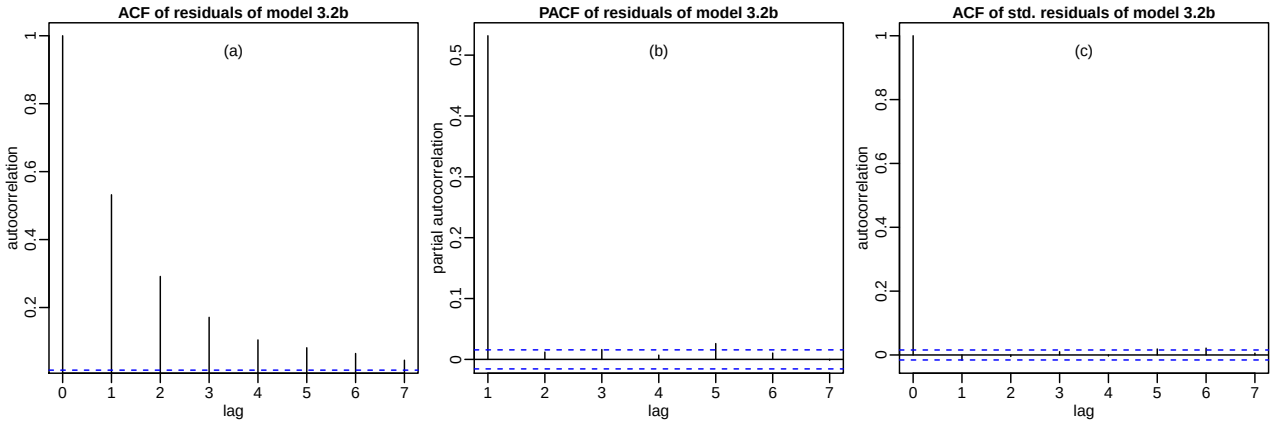


Figure 3.7: The AR(1) error assumption for model 3.2b is adequate. Although raw residuals are still AR(1) correlated (see panel (a) and panel (b)), standardized residauls are now i.i.d. (see panel (c)). See §2.2.2 if you need a revision on residual standardization.

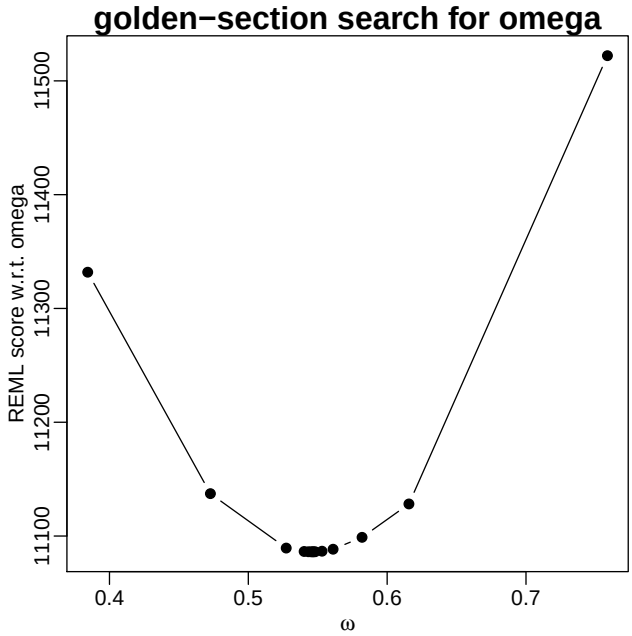


Figure 3.8: The search path of golden-section iterations for fitted model 3.2b. A point estimate of $\hat{\omega} = 0.5447645$ is obtained after 14 iterations.

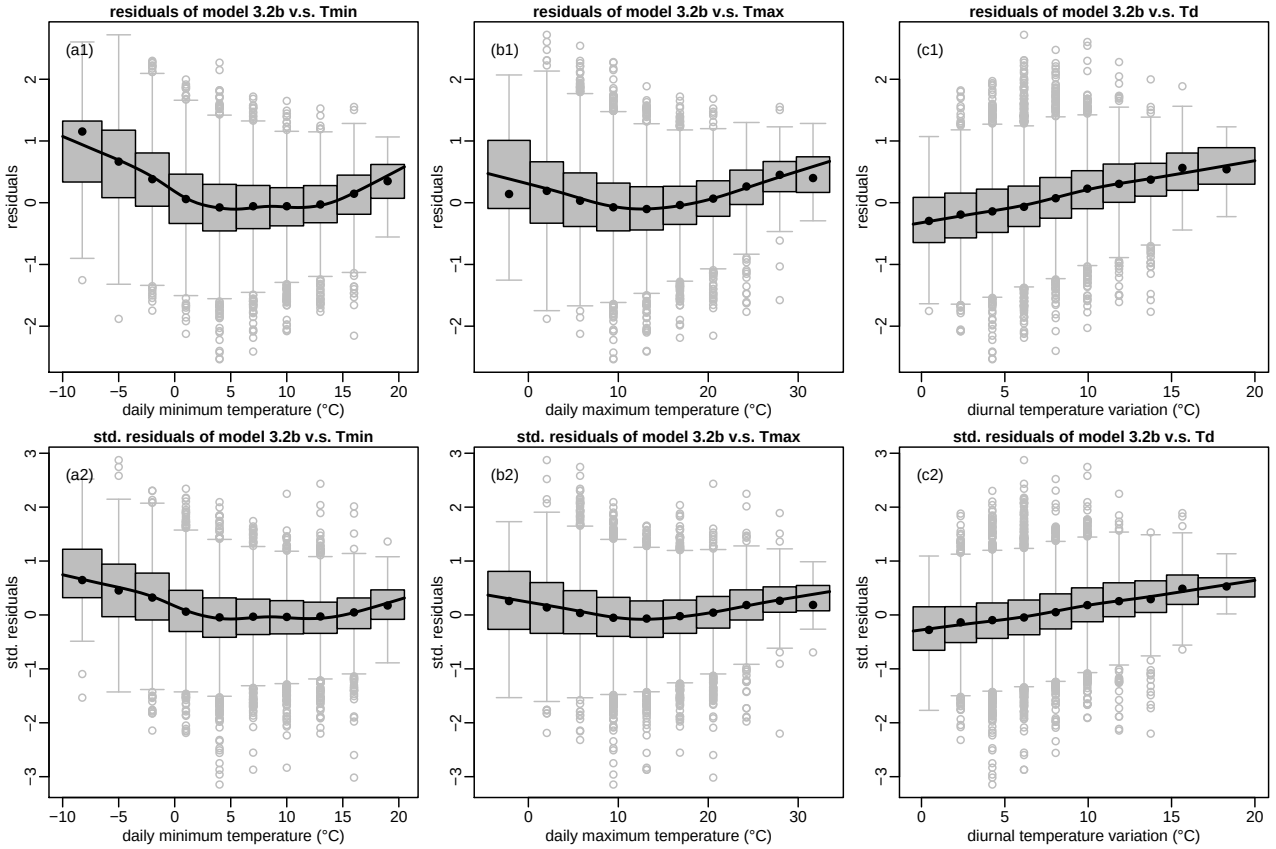


Figure 3.9: Inspecting residuals of fitted model 3.2b agains daily minimum temperature (Tmin) $\mathtt{T}^0_\mathtt{d}$, daily maximum temperature (Tmax) $\mathtt{T}^1_\mathtt{d}$ and diurnal temperature variation (Td) $\mathtt{T}^*_\mathtt{d}$. Unmodelled trend has been detected so these covariates could improve our model. Since $\mathtt{T}^0_\mathtt{d}$ and $\mathtt{T}^1_\mathtt{d}$ are highly correlated, they should not enter the model in an additive manner; they need be jointly modelled as a bivariate thin-plate spline. An alternative idea may be including $\mathtt{T}^0_\mathtt{d}$ and $\mathtt{T}^*_\mathtt{d}$ in an additive manner as these two variables are uncorrelated. The plotting method used is explained in §2.4.

3.2.4 Including meteorological covariates

So far no meteorological covariates have been used for model building. It is time to see if they can improve our model. For this *Manchester 11* case study, I will simply denote temperature variables by for example, $\mathtt{T}^0_\mathtt{d}$ than $\mathtt{T}^0_\mathtt{id}$, because there is only a single station.

Figure 3.9 sketches residuals and standardized residuals of fitted model 3.2b against daily minimum temperature $\mathtt{T}^0_\mathtt{d}$, daily maximum temperature $\mathtt{T}^1_\mathtt{d}$ and diurnal temperature variation $\mathtt{T}^*_\mathtt{d}$. A clear, similar non-linear relationship is seen between residuals and $\mathtt{T}^0_\mathtt{d}$ or $\mathtt{T}^1_\mathtt{d}$. This is probably because that $\mathtt{T}^0_\mathtt{d}$ and $\mathtt{T}^1_\mathtt{d}$ are highly correlated (their Pearson correlation coefficient is 0.82 at *Manchester 11*). This means that if both variables would enter the model, they should not be included in an additive manner. A bivariate thin-plate spline is more appropriate. One motivation of this bivariate function is that interaction between $\mathtt{T}^0_\mathtt{d}$ or $\mathtt{T}^1_\mathtt{d}$ can also be modelled. Such interaction may make sense. Suppose that two days have the same daily minimum temperature at -5°C, but one day has a daily maximum temperature at 10°C while the other has that at 0°C. Obviously the second day is colder so there is probably more heating supply, more fuel combustion hence more Smoke emission. In other words, $\mathtt{T}^0_\mathtt{d}$ alone may not be fully explain the temperature effect on Smoke concentration; diurnal temperature variation could matter as well. Panel (c1) and panel (c2) of Figure 3.9 seem to verify this. In fact, $\mathtt{T}^0_\mathtt{d}$ and $\mathtt{T}^*_\mathtt{d}$ can be included in a model in an additive manner because they are basically uncorrelated (their Pearson correlation coefficient is 0.02 at *Manchester 11*). I thus consider a model

Table 3.5: Effective degree of freedom for all functions in fitted model 3.3a, assuming AR(1) error with unknown autocorrelation coefficient.

$\hat{f}_1$	$\hat{f}_{1,2}$	$\hat{f}_{1,3}$	$\hat{f}_{1,4}$	$\hat{f}_{1,5}$	$\hat{f}_{1,6}$	$\hat{f}_{1,7}$
9.50	0.00	0.00	0.00	1.14	1.33	2.12
$\hat{f}_2$	$\hat{f}_{2,2}$	$\hat{f}_{2,3}$	$\hat{f}_{2,4}$	$\hat{f}_{2,5}$	$\hat{f}_{2,6}$	$\hat{f}_{2,7}$
20.80	1.79	2.89	3.77	2.45	1.00	8.43
$\hat{f}_3$	$\hat{f}_4$	$\hat{f}_5$				
87.46	8.57	3.74				

Model 3.3a: model 3.2b + f_4 + f_5

$$\begin{aligned}\texttt{logBS}_\texttt{d} = u_1 + \sum_{s=2}^{7} \delta_s^{\texttt{d}_\texttt{w}} u_s + f_1(\texttt{w}_\texttt{y};\ 15) + \sum_{s=2}^{7} \delta_s^{\texttt{d}_\texttt{w}} f_{1,s}(\texttt{w}_\texttt{y};\ 15) + \\ f_2(\texttt{w};\ 40) + \sum_{s=2}^{7} \delta_s^{\texttt{d}_\texttt{w}} f_{2,s}(\texttt{w};\ 40) + f_3(\texttt{w}_\texttt{y},\ \texttt{w};\ 10,\ 40) + \\ f_4(\texttt{T}_\texttt{d}^0;\ 15) + f_5(\texttt{T}_\texttt{d}^*;\ 10) + e_\texttt{d},\end{aligned}$$

where new components to model 3.2b are

1. f_4 is a natural cubic spline of $\texttt{T}_\texttt{d}^0$, with 15 knots (this seems more than sufficient from Figure 3.9);
2. f_5 is a natural cubic spline of $\texttt{T}_\texttt{d}^*$, with 10 knots (this seems more than sufficient from Figure 3.9; note that while the relationship appears linear from Figure 3.9, it might still be a good idea to model it as a spline, as such relationship can be non-linear for other stations than *Manchester 11* and we should not overly restrict the form representation at this stage);
3. $e_\texttt{d}$ is an AR(1) model error with unknown autocorrelation coefficient.

Fitting this model proves temperature variables extremely useful. They not only substantially reduces the estimated residual variance (model 3.2b has residual variance 0.3321138, while model 3.3a has residual variance 0.2799806), but also slightly decreases the autocorrelation in residuals (model 3.2b has estimated AR(1) correlation at 0.5447645, while the point estimate for model 3.2b is 0.517817). This is probably because that daily temperature is also a time series with strong autocorrelation, which can explain some autocorrelation in logBS as well. (Later in Figure 3.20 I will highlight how effectively temperature variables can improve model prediction.) Table 3.5 summarizes the effective degree of freedom for all functions in the model (note that f_5 ends up with a value of 3.75 so it is slightly nonlinear!). Figure 3.10 once again inspects model residuals against temperature variables. It is interesting to notice that while daily maximum temperature has not been used for building model, the previously spotted unmodelled trend in panel (b1) and panel (b2) or Figure 3.9 has largely gone!

It is less easy to judge whether there is an interaction between $\texttt{T}_\texttt{d}^0$ and $\texttt{T}_\texttt{d}^*$, that is, should the following model be considered,

Model 3.3b: model 3.3a + f_6

$$\begin{aligned}\texttt{logBS}_\texttt{d} = u_1 + \sum_{s=2}^{7} \delta_s^{\texttt{d}_\texttt{w}} u_s + f_1(\texttt{w}_\texttt{y};\ 15) + \sum_{s=2}^{7} \delta_s^{\texttt{d}_\texttt{w}} f_{1,s}(\texttt{w}_\texttt{y};\ 15) + \\ f_2(\texttt{w};\ 40) + \sum_{s=2}^{7} \delta_s^{\texttt{d}_\texttt{w}} f_{2,s}(\texttt{w};\ 40) + f_3(\texttt{w}_\texttt{y},\ \texttt{w};\ 10,\ 40) + \\ f_4(\texttt{T}_\texttt{d}^0;\ 15) + f_5(\texttt{T}_\texttt{d}^*;\ 10) + f_6(\texttt{T}_\texttt{d}^0,\ \texttt{T}_\texttt{d}^*;\ 15,\ 10) + e_\texttt{d},\end{aligned}$$

where f_6 is a tensor product spline between a natural cubic spline of $\texttt{T}_\texttt{d}^0$ and a natural cubic spline of $\texttt{T}_\texttt{d}^*$? Actually fitting this model shows that f_6 has a degree of freedom at 20.11 (see Table 3.6)

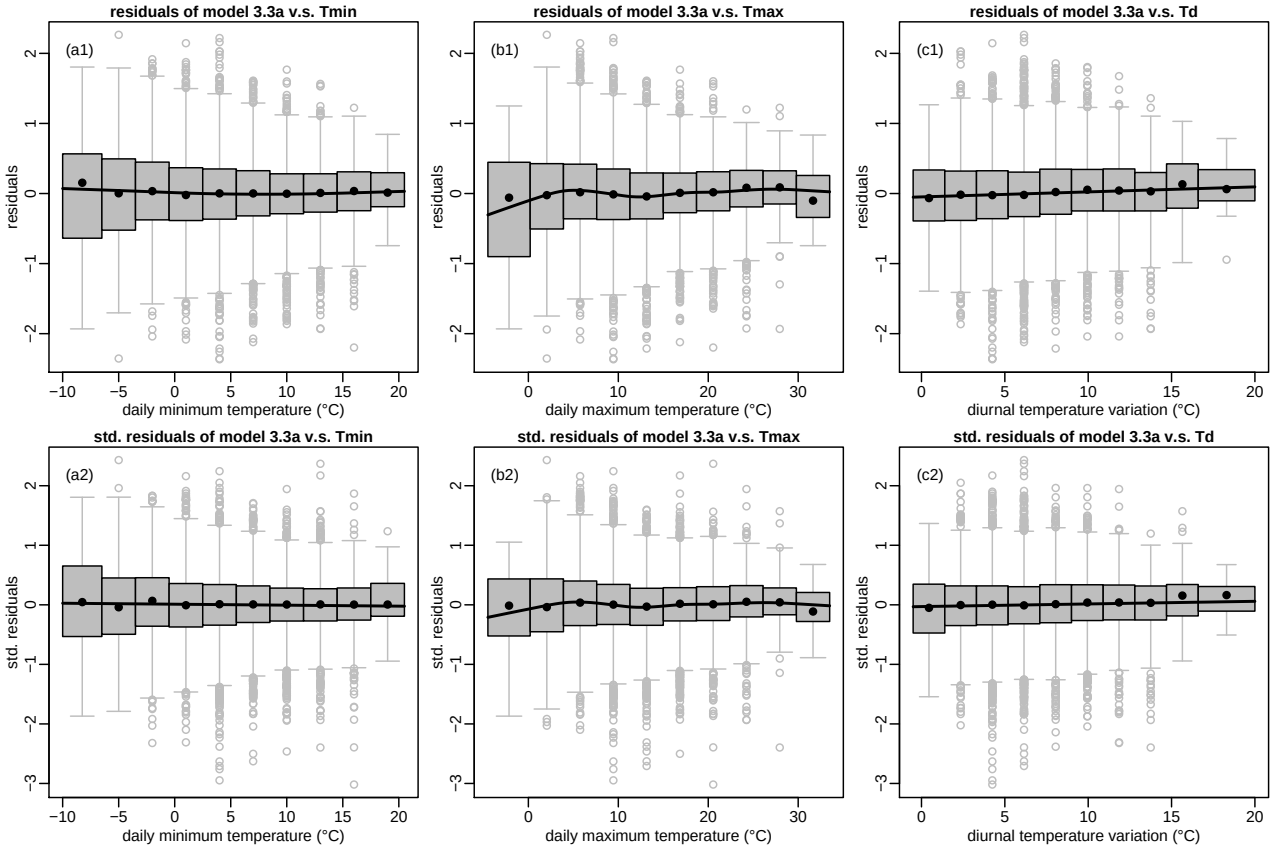


Figure 3.10: Inspecting residuals of fitted model 3.3a agains daily minimum temperature (Tmin) $\mathtt{T}_\mathtt{d}^0$, daily maximum temperature (Tmax) $\mathtt{T}_\mathtt{d}^1$ and diurnal temperature variation (Td) $\mathtt{T}_\mathtt{d}^*$. Unmodelled trend w.r.t. $\mathtt{T}_\mathtt{d}^0$ and $\mathtt{T}_\mathtt{d}^1$ which are previously spotted in Figure 3.9 for model 3.2b have been removed. The unmodelled trend w.r.t. $\mathtt{T}_\mathtt{d}^1$ has also been largely taken off, although this variable is not used for building model 3.3a.

Table 3.6: Effective degree of freedom for all functions in fitted model 3.3b, assuming AR(1) error with unknown autocorrelation coefficient.

$\hat{f}_1$	$\hat{f}_{1,2}$	$\hat{f}_{1,3}$	$\hat{f}_{1,4}$	$\hat{f}_{1,5}$	$\hat{f}_{1,6}$	$\hat{f}_{1,7}$
9.69	0.00	0.00	0.00	1.10	1.33	2.17
$\hat{f}_2$	$\hat{f}_{2,2}$	$\hat{f}_{2,3}$	$\hat{f}_{2,4}$	$\hat{f}_{2,5}$	$\hat{f}_{2,6}$	$\hat{f}_{2,7}$
20.83	1.71	2.90	3.73	2.47	1.00	8.45
$\hat{f}_3$	$\hat{f}_4$	$\hat{f}_5$	$\hat{f}_6$			
86.85	7.67	1.00	20.11			

which is relatively high. Perhaps some idea can be obtained by surface visualization. Figure 3.11 is a perspective plot for $\hat{f}_6$; it does appear that there is nonlinear interaction between $\mathtt{T}_\mathtt{d}^0$ and $\mathtt{T}_\mathtt{d}^*$. Figure 3.12 compares surface of $\hat{f}_4 + \hat{f}_5$ and surface of $\hat{f}_4 + \hat{f}_5 + \hat{f}_6$. The interaction does add some extra curvature. At this stage it is probably a good idea not to overly restrict the model structure, as eventually we need to build models for the whole monitoring network. So I will choose to retain this interaction term in the model for now.

The UKCP09 dataset also provides rainfall data but only at a monthly resolution. It is unclear whether a monthly covariate is helpful in explaining daily logBS. Figure 3.13 sketches residuals and standardized residuals of fitted model 3.3b against monthly rainfall. It appears that at least at station *Manchester 11*, there isn't strong relationship between rainfall and logBS. However, this can not exclude the impact of rainfall in general. Notably, the rainfall measurements at this particular station only vary between 1 and 227, however, rainfall measurements at all stations of the Network vary between 1 and 583 which is a much wider range. So the effect of rainfall needs be investigated at a later stage.

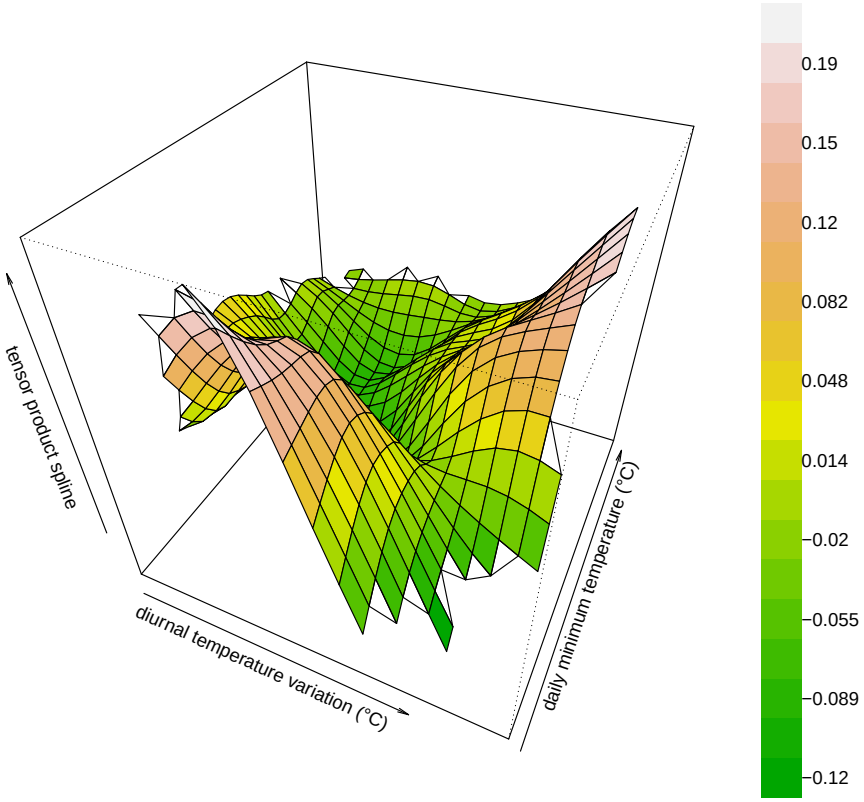
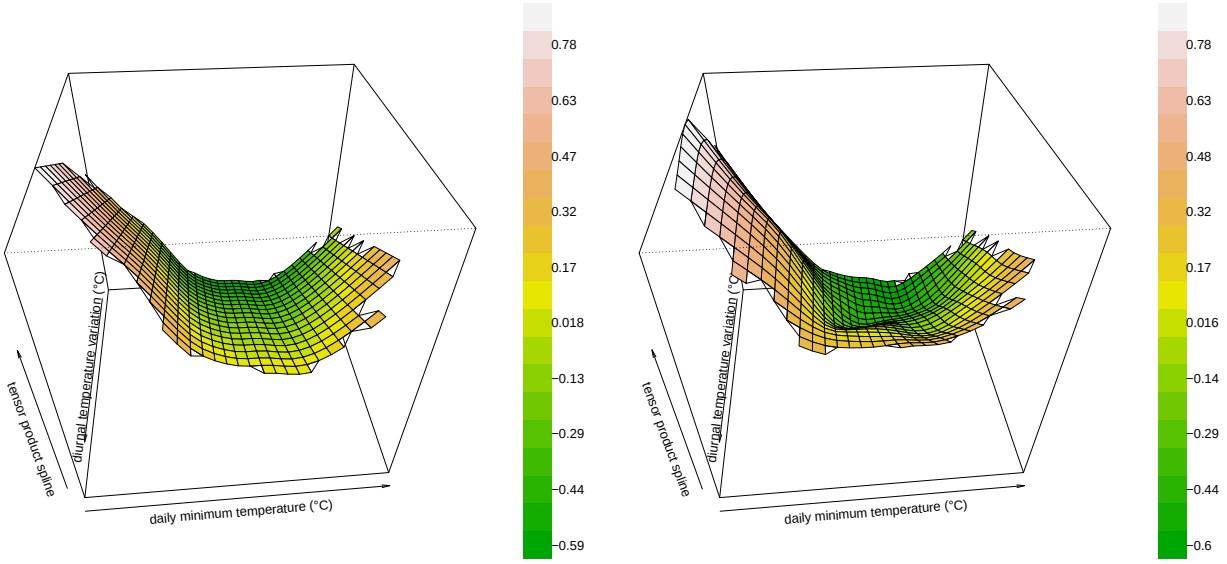


Figure 3.11: Perspective plot for $\hat{f}_6$ in fitted model 3.2b. The surface is highly nonlinear, so there is probably a notable interaction between $\mathtt{T}_\mathtt{d}^0$ and $\mathtt{T}_\mathtt{d}^*$. See §2.3 for more details on such kind of plot.



(a) $\hat{f}_4 + \hat{f}_5$ **(b)** $\hat{f}_4 + \hat{f}_5 + \hat{f}_6$

Figure 3.12: Comparing $\hat{f}_4 + \hat{f}_5$ in the fitted model 3.3a and $\hat{f}_4 + \hat{f}_5 + \hat{f}_6$ in the fitted model 3.3b. The interaction adds some extra, noticable curvature to the surface.

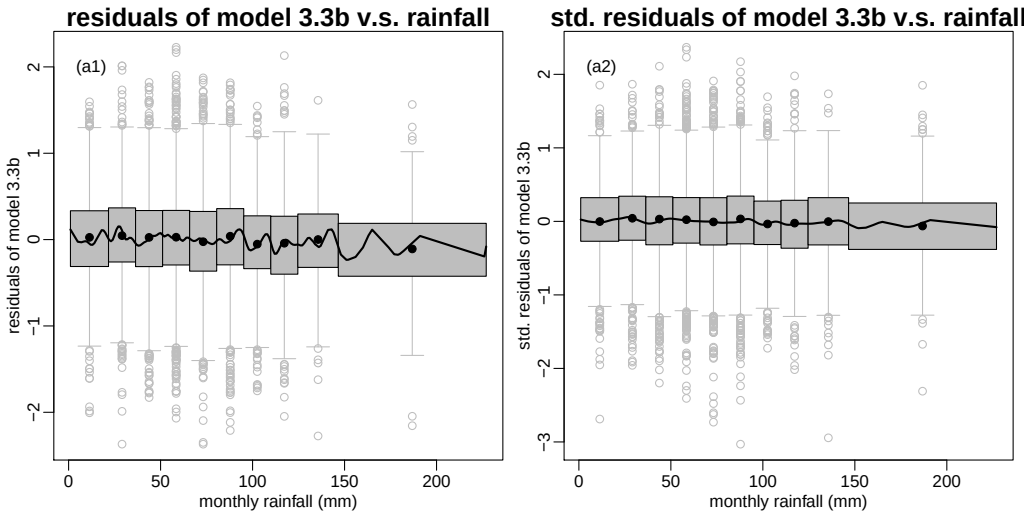


Figure 3.13: Inspecting residuals and standardized residuals of fitted model 3.3b against monthly rainfall. It appears that at station *Manchester 11*, there isn't strong relationship between rainfall and logBS. However, this can not exclude the impact of rainfall in general. Notably, the rainfall measurements at this particular station only vary between 1 and 227, however, rainfall measurements at all stations of the Network vary between 1 and 583 which is a much wider range. So the effect of rainfall needs be investigated at a later stage.

3.2.5 Model checking and visualization

The fitted model 3.3b involves 732 coefficients for 15707 data, with a resulting degree of freedom at 208.64. The estimated residual variance is 0.2779, while the sample variance of logBS observations is 1.1144, so the the fitted model has an adjusted R-squared of 1 - 0.2779 / 1.1144 = 75.06%. I will now perform some model checking for this model and visualize all spline functions.

Figure 3.14 conducts some residual checking. Panels (a) to (c) just reassure that autocorrelation in daily logBS has been adequately modelled by an AR(1) process. Panel (e) for residuals v.s. fitted values might cause some concern. However, the increase at the right boundary really implies that a smoothing model has difficulty in approximating unextraordinarily large observations (extreme values). The highest value observed at *Manchester 11* is 8.6 for logBS, but 5413 μgm^{-3} in the original scale (even larger than the 4500 μgm^{-3} in London's 1952 episode of fog!). The fitted model predicts 6.36 for logBS and the residual is as high as 2.23. In general, the constant error variance assumption is not violated.

Figure 3.15 is an illustration of the estimated long term trend $\hat{f}_2$, within-year seasonality $\hat{f}_1$ and weekly effect $\hat{u}_1$, $\hat{u}_1 + \hat{u}_2$, ..., $\hat{u}_1 + \hat{u}_7$. There are a few interesting observations.

1. The long term trend follows a general decline, except for some time around week 1500 (around mid 1980's).
2. The "V" shape curve of seasonal pattern is not surprising, as winter is when heating demands and fuel combustion are high. However, the peak of the curve is somewhere around November not December. Perhaps people went on holiday near Christmas and there is less industrial and commercial activity than usual?
3. The weekly pattern suggests that Sunday is normally when Smoke concentration is the lowest in a week. However, the difference between Saturday and Sunday is rather notable. Also, while Tuesday, Wednesday, Thursday and Friday are more alike, Monday is quite different. The reason behind this phenomenon is unknown, but it is readily clear that it is not a good idea to simply model weekly effect as a two-level factor for weekday / weekend.

The most obvious way to visualize interaction term $\hat{f}_{1,2}$, ..., $\hat{f}_{1,3}$ is to plot them individually against

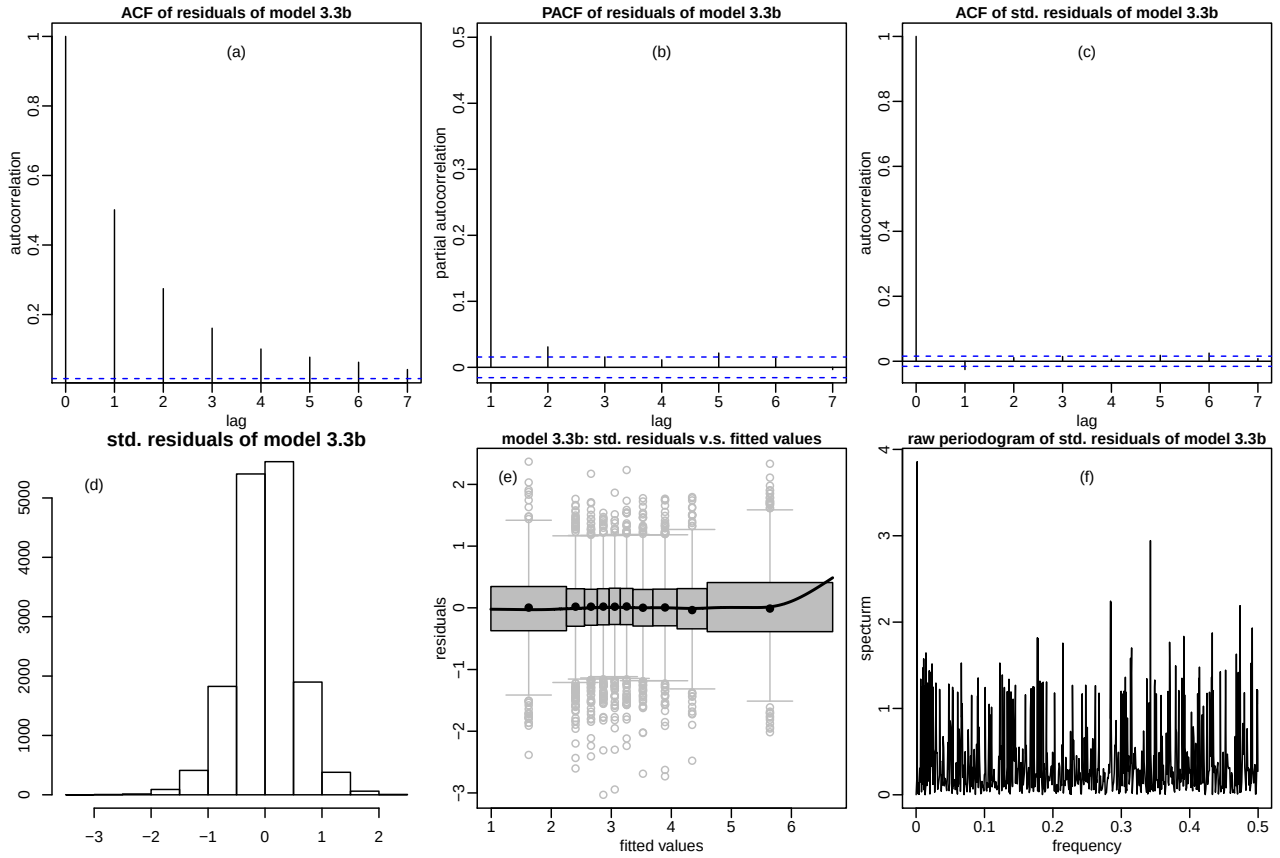


Figure 3.14: Residual checking for fitted model 3.3b. Panel (a) is the ACF for residuals. Panel (b) is the PACF for residuals. Panel (c) is the ACF for AR(1) standardized residuals. These panels convince that autocorrelation in daily logBS has been well modelled. Panel (d) is a histogram of standardized residuals. Panel (e) sketches standardized residuals against fitted values. Generally the constant variance assumption for model error is not violated. The increase at the right boundary implies that a smoothing model has difficulty in approximating unextraordinarily large observations (extreme values). The highest value observed at *Manchester 11* is 8.6 for logBS, but 5413 μgm^{-3} in the original scale. The fitted model predicts 6.36 for logBS and the residual is as high as 2.23. Panel (f) is a raw periodgram for standardized residuals. It verifies that they are almost white noise without any unmodelled cyclic / periodic effect.

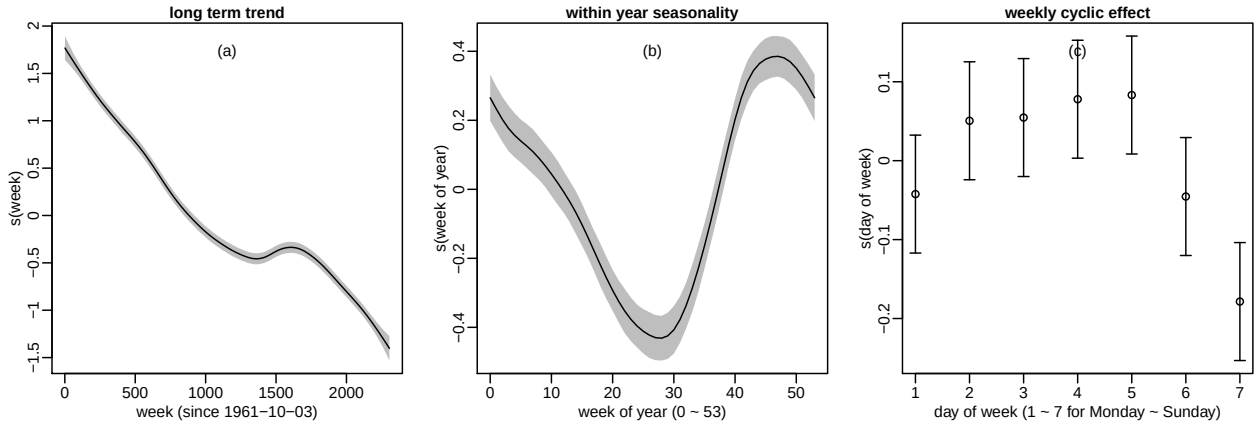


Figure 3.15: Illustration of the general long term trend $\hat{f}_2$, within-year seasonality $\hat{f}_1$ and weekly effect $\hat{u}_1$, $\hat{u}_1 + \hat{u}_2$, ..., $\hat{u}_1 + \hat{u}_7$ in fitted model 3.3b. The gray ribbon covers region that is within 2 standard deviation of the estimate.

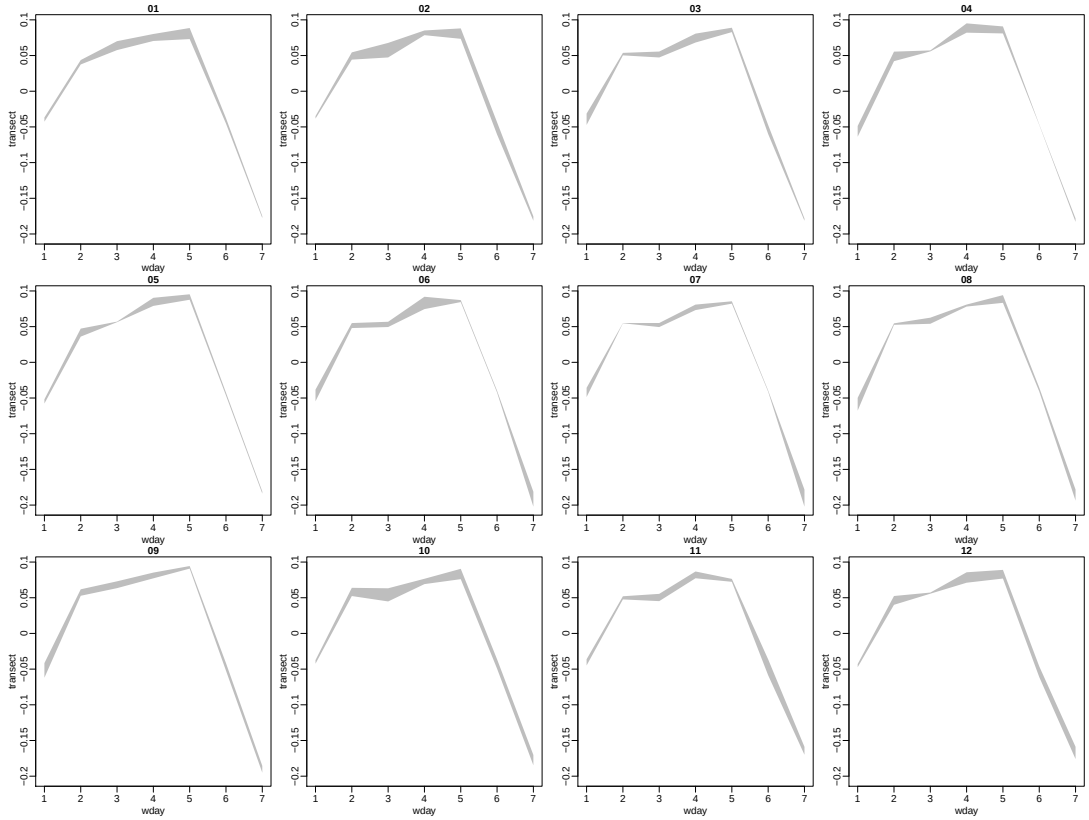


Figure 3.16: Grouped transect plots (by month) for $\hat{f}_1$, $\hat{f}_1 + \hat{f}_{1,2}$, ..., $\hat{f}_1 + \hat{f}_{1,7}$ in the fitted model 3.3b. There is plenty of **cyclic** variation of weekly pattern with a year, notably for the middle of a week (i.e., Tuesday to Friday).

$\mathtt{w_y}$. Similarly $\hat{f}_{2,2}$, ..., $\hat{f}_{2,7}$ can be each plotted against $\mathtt{w}$. However, it is probably more informative to learn how the weekly pattern varies within a year and changes in a long term. For this purpose, I will for example, fix $\mathtt{w_y}$ at a specific value and plot the resulting 7 values $\hat{f}_1(\mathtt{w_y})$, $\hat{f}_1(\mathtt{w_y}) + \hat{f}_{1,2}(\mathtt{w_y})$, ..., $\hat{f}_1(\mathtt{w_y}) + \hat{f}_{1,7}(\mathtt{w_y})$. Yet this is not ideal as this single line does not express the local variation along the other coordinate $\mathtt{w_y}$. A better option is to group transects and plot them in chunks. For example, for $\hat{f}_1(\mathtt{w_y})$, $\hat{f}_1(\mathtt{w_y}) + \hat{f}_{1,2}(\mathtt{w_y})$, ..., $\hat{f}_1(\mathtt{w_y}) + \hat{f}_{1,7}(\mathtt{w_y})$ I will group weeks of year into 12 months and produce 12 grouped transect plots. For $\hat{f}_2(\mathtt{w})$, $\hat{f}_2(\mathtt{w}) + \hat{f}_{2,2}(\mathtt{w})$, ..., $\hat{f}_2(\mathtt{w}) + \hat{f}_{2,7}(\mathtt{w})$ I will group weeks into 45 years and produce 45 grouped transect plots. For $\hat{f}_3(\mathtt{w_y},\ \mathtt{w})$, the interest is how seasonal pattern would vary in a long term and the idea of grouped transect plots applies as well. Figure 3.16, Figure 3.17 and Figure 3.18 illustrate such grouped transect plots for these model components.

1. From Figure 3.16 we see that there is plenty of **cyclic** variation of weekly pattern within a year (notably for the middle of a week, i.e., Tuesday to Friday), that is, these changes will be replicated for all 45 years. .

2. From Figure 3.17 we see that there isn't much **non-cyclic** variation of weekly pattern within a year (the gray ribbon contains 53 transect lines, but they are pretty much clustered together; so this can probably be interpreted that all variation of weekly pattern in a year is cyclic, as is captured by $\hat{f}_{1,2}(\mathtt{w_y},\ \mathtt{w})$ to $\hat{f}_{1,7}(\mathtt{w_y},\ \mathtt{w})$), but the long term change is dramatic.

3. From Figure 3.18 we see that the seasonal patter can vary greatly within a year. This needs some clarification. Isn't a year a period of seasonal pattern? How can it vary? Well, the general seasonality $\hat{f}_1$ is static, but it does not mean that it is simply replicated for all 45 years, or only rescaled once per year; its shape can change smoothly as each day passes. So this tensor product spline is capable to model very complicated dynamics.

In the previous section we have visualized the interaction term $\hat{f}_6$ for $\mathtt{T_d^0}$ and $\mathtt{T_d^*}$. Their marginal effects are now illustrated in Figure 3.19. There should be some more sophisticated scientific explanation on

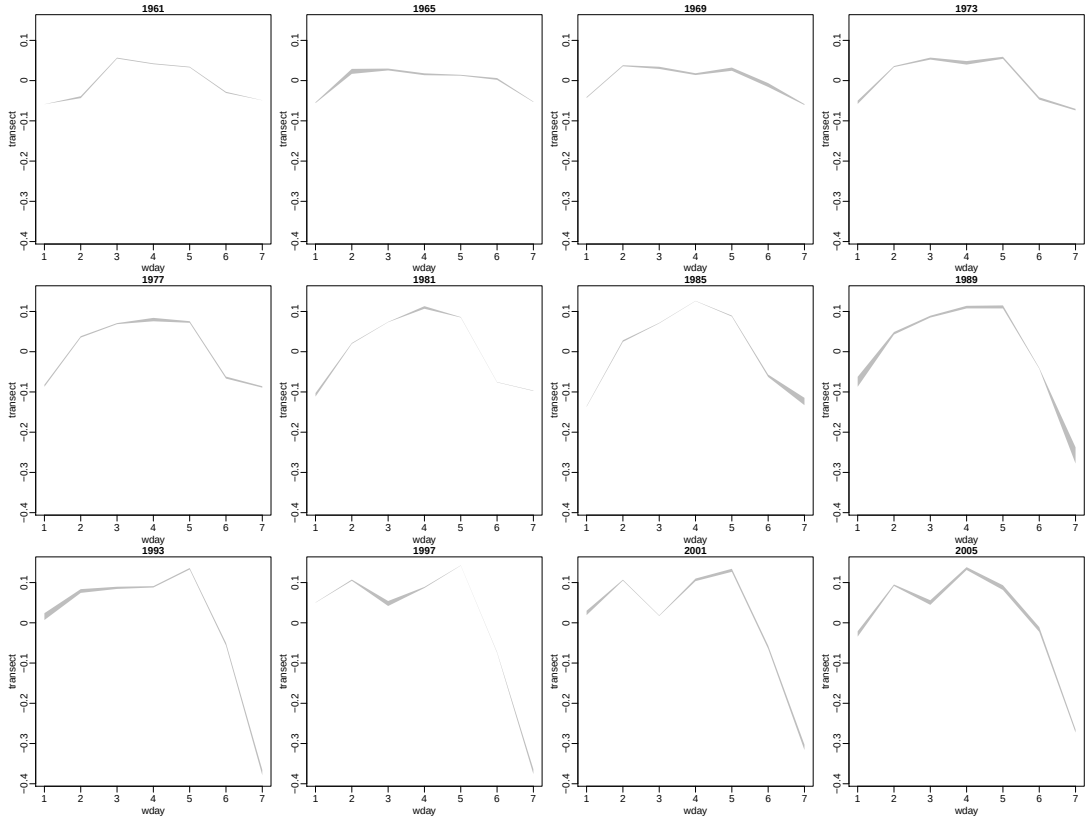


Figure 3.17: Grouped transect plots (by year) for $\hat{f}_2$, $\hat{f}_2 + \hat{f}_{2,2}$, ..., $\hat{f}_2 + \hat{f}_{2,7}$ in the fitted model 3.3b. There isn't much **non-cyclic** variation of weekly pattern within a year (the gray ribbon contains 53 transect lines, but they are pretty much clustered together), but the long term change is dramatic. Note that for space reason, the plot is only made every 4 years.

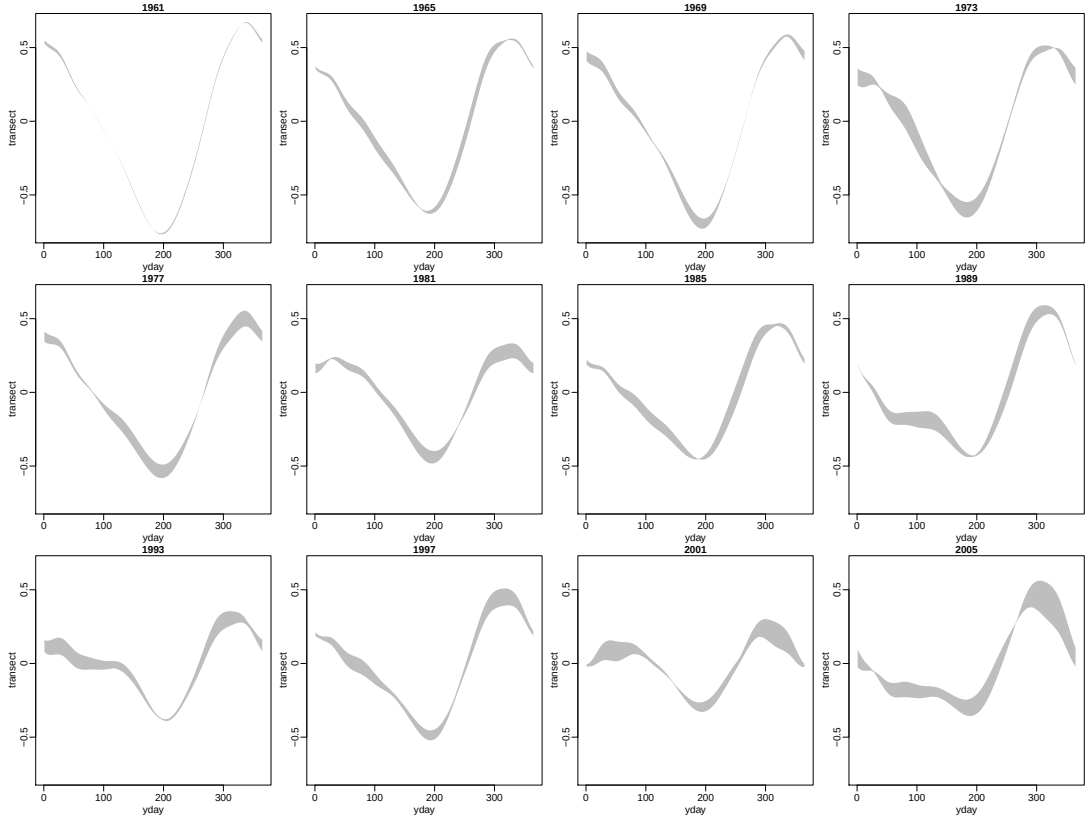


Figure 3.18: Grouped transect plots (by year) for $\hat{f}_1(\mathtt{w_y}) + \hat{f}_3(\mathtt{w_y},\ \mathtt{w})$ in the fitted model 3.3b. The wide width of the gray ribbon implies that the seasonal patter vary greatly within a year. For space reason, the plot is only presented every 4 years.

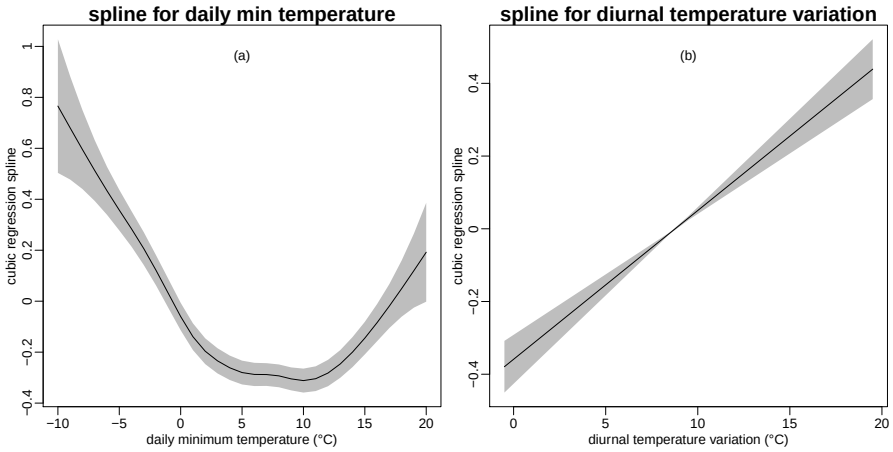


Figure 3.19: Marginal effects of daily minimum temperature ($\hat{f}_4$) and diurnal temperature variation ($\hat{f}_5$) in fitted model 3.3b. There should be some more sophisticated (and interesting) scientific explanation on the impact of temperature on Smoke concentration, but it is out of scope of this PhD thesis.

the impact of temperature on Smoke concentration, other than the association between temperature and heating. For example, the positive slope of $\mathtt{T}^*_\mathtt{d}$ is not something that can be explained by heating. However, this scientific topic is out of scope of this PhD thesis.

After visualizing each individual splines, let us see how they help prediction when added together. Figure 3.20 breaks down components of fitted model 3.3b into trend, seasonal, weekly and temperature components and illustrates their contribution to logBS prediction in four decades. The weekly component have greater and greater effects in later decades, and the temperature components substantially improve the quality of prediction.

3.2.6 Summary

The model development for a logBS time series has been very much detailed. Before moving on, I will summarize key messages that can be learnt from this *Manchester 11* case study.

First of all,

- A logBS daily time series can be decomposed into additive smooth functions of time variables at different resolutions (as well as their interactions), namely week $\mathtt{w}$, day of week $\mathtt{d_w}$ and week of year $\mathtt{w_y}$, plus an AR(1) process mainly for autocorrelation with a week.

In addition, we have obtained some rough idea of the complexity for each component:

- Conditional on an adequate AR(1) model for autocorrelation in daily data, $15 \sim 20$ knots should be sufficient for variable $\mathtt{w_y}$, and $40 \sim 50$ knots should be more than sufficient for variable $\mathtt{w}$. Temperature variables do not need very high k values; 15 knots should be more than sufficient for both $\mathtt{T}^0_\mathtt{d}$ and $\mathtt{T}^*_\mathtt{d}$.

Of course, it would be good if such rough idea drawn from *Manchester 11* case study can be validated on other stations.

As one way to do this, let us consider a subset of the full logBS dataset, where only stations with long monitoring history, say over 10000 days' records, are retained. It turns out that 69 stations are

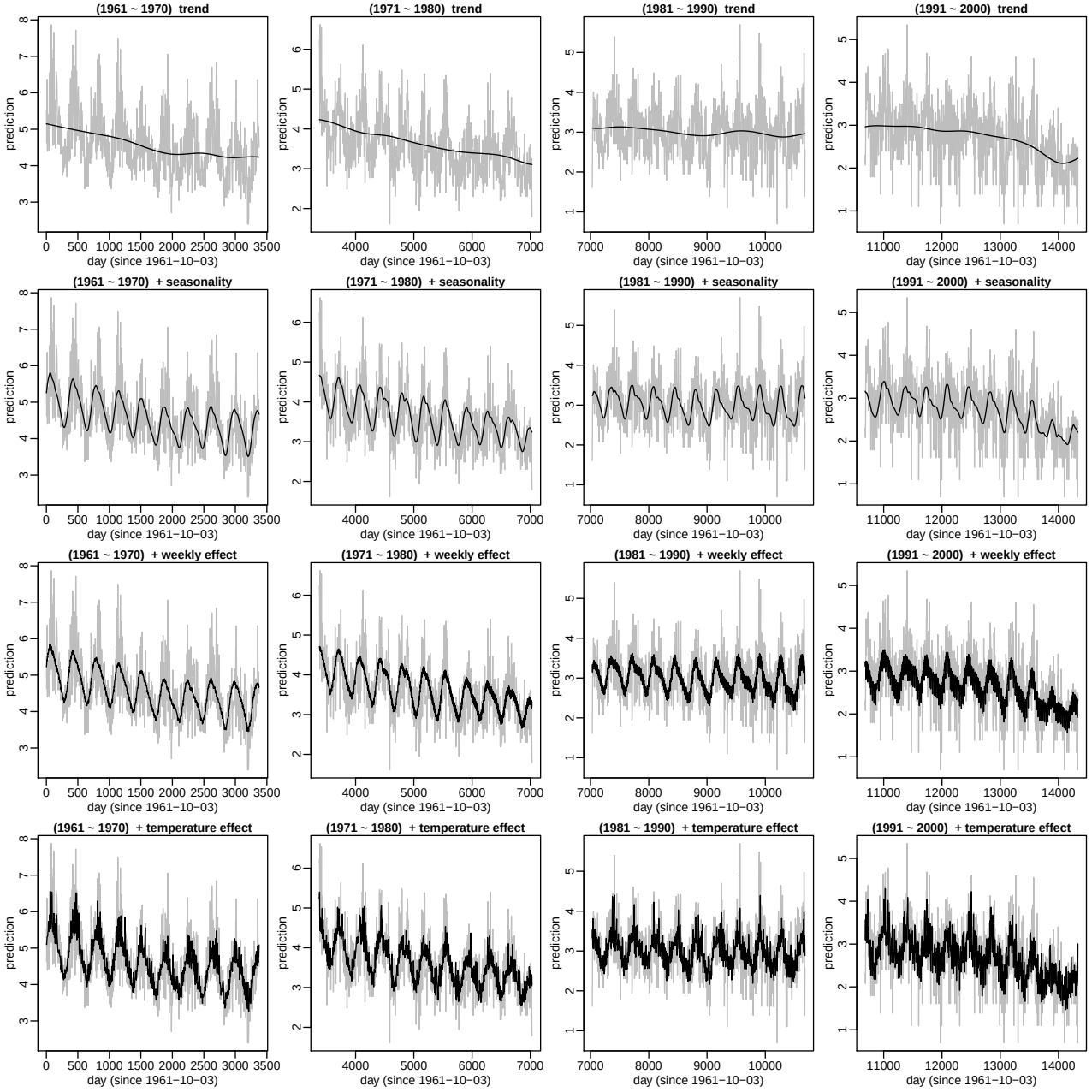


Figure 3.20: An illustration of how the fitted model 3.3b predicts logBS when its model terms are added together. The daily time series is broken into four decades, each per column panel. For column panels: the topmost row shows trend component $\hat{u}_1 + \hat{f}_2$ only; then the second row adds seasonal components $\hat{f}_1$ and $\hat{f}_3$; the third row further adds weekly components $\hat{u}_2$, ..., $\hat{u}_7$, $\hat{f}_{1,2}$, ..., $\hat{f}_{1,7}$ and $\hat{f}_{2,1}$, ..., $\hat{f}_{2,7}$; finally the bottom row adds temperature components $\hat{f}_4$, $\hat{f}_5$ and $\hat{f}_6$. The weekly components have increasingly noticeable effects as time goes by, and the temperature components substantially improve prediction.

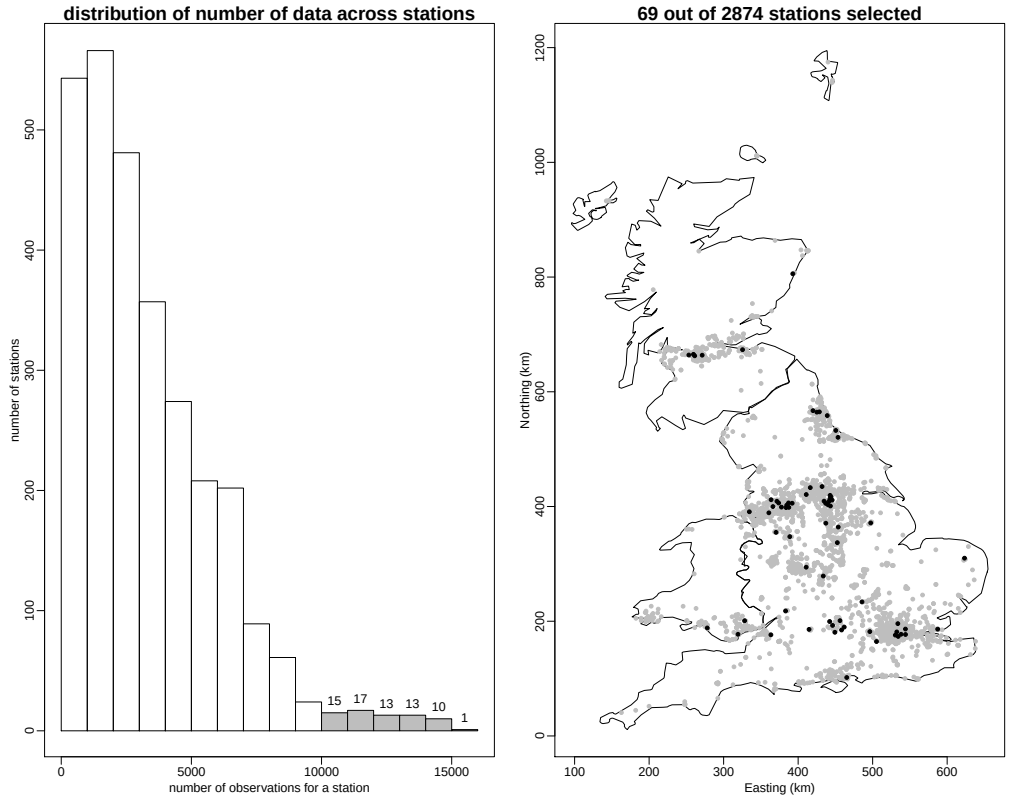


Figure 3.21: All 69 stations with over 10000 days' measurements are selected to validate my rough estimate on the number of knots for various splines in model 3.3b. The panel on the left is a histogram for the number of measurements per station. Only a small proportion of stations have a long monitoring history (10000 days $\approx$ 30 years). The right panel identifies those stations on the map. From their relative positions (black dots) to all other historical stations (gray dots) in the Network, they seem to be pretty representative of the Network. These stations contribute to about 9% of the all daily measurements from the Network.

selected (see Figure 3.21). From their relative positions to all other stations in the Network, they seem to be pretty representative of the Network. Dropping weekly components from model 3.3b, we can fit the reduced model for logBS on each day of week per station (assuming i.i.d. errors of course). This ends up with $69 \times 7 = 483$ models. Repeating the "choose k" procedure for all of them, I summarize the result in Figure 3.22. The Figure is produced in a similar way to Figure 3.4, except that the line plot is replaced by their bounding polygon. The effective degree of freedom plateaus for all splines as k increases, and except that I might have slighly underestimated the complexity required for $f_2(\mathtt{w})$, the estimates for all terms are fairly adequate.

Another useful message is that meteorological variables, particularly temperature variables $\mathtt{T}_\mathtt{d}^0$ and $\mathtt{T}_\mathtt{d}^*$, are extremely helpful for prediction. However, from the *Manchester 11* dataset (too small a sample size) the following can not be learned so they need be revisited in future.

1. Is the temperature effect variable in a long term, that is, is there an iteraction between temperature and year $\mathtt{y}$?
2. Will rainfall affect logBS?

There are other alternative ways to model temperature effects. Using a bivariate thin-plate spline for $\mathtt{T}_\mathtt{d}^0$ and $\mathtt{T}_\mathtt{d}^*$ turns out to produce the same surface as is graphed in Figure 3.12. In fact, it is also equivalent to use a bivariate thin-plate spline for $\mathtt{T}_\mathtt{d}^0$ and $\mathtt{T}_\mathtt{d}^1$, although the surface lies in a different domain (see Figure 3.23). The thin-plate spline representation is very concise: we can just write one function instead of three for temperature variables. However, it is probably preferable to use the tensor product spline representation: $f_4(\mathtt{T}_\mathtt{d}^0) + f_5(\mathtt{T}_\mathtt{d}^*) + f_6(\mathtt{T}_\mathtt{d}^0,\ \mathtt{T}_\mathtt{d}^*)$ where the marginal effects f_4 and f_5 can be visualized (see Figure 3.19).

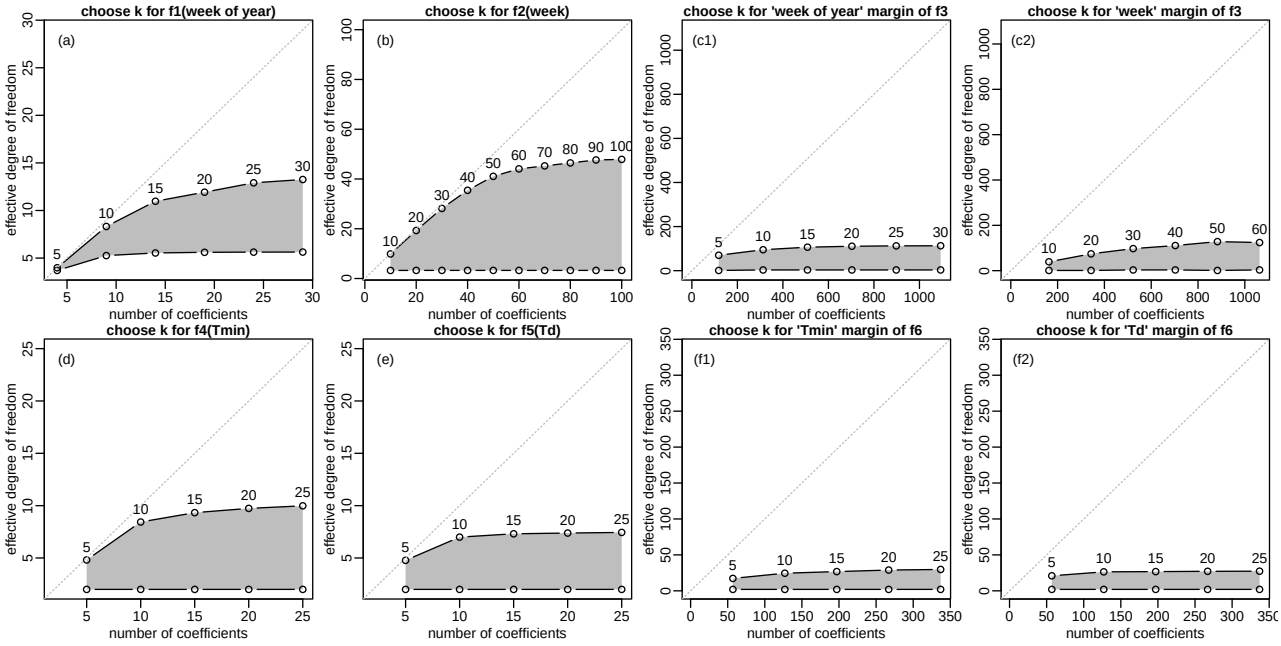


Figure 3.22: The result of "choose k" process for fitting model 3.3b (replacing `d` with `v`, dropping weekly components and hence assuming i.i.d. errors) for logBS on each day of week per station. This gives a good estimate on adequate k values for the model.

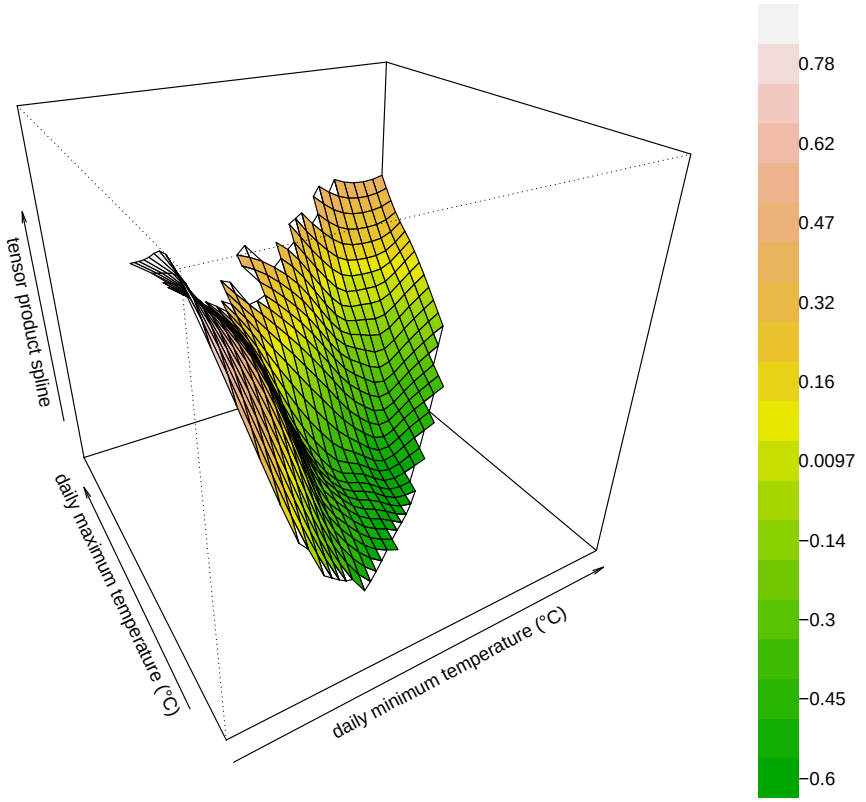


Figure 3.23: Using a bivariate thin-plate spline for $\mathtt{T}^0_\mathtt{d}$ and $\mathtt{T}^*_\mathtt{d}$ produces a surface on a different domain. Since these two variables are correlelated the domain stretches along the diagonal of the square. But after a transformation $\mathtt{T}^0_\mathtt{d} = \mathtt{T}^0_\mathtt{d}$ and $\mathtt{T}^*_\mathtt{d} = \mathtt{T}^1_\mathtt{d} - \mathtt{T}^0_\mathtt{d}$ the surface almost coincides the surface in Figure 3.12.

There are also other ways to introduce $\texttt{d}_\texttt{w}$ into the model than using a factor and replicating the smooth function for each level. These alternatives include:

- using a cubic cyclic spline for $\texttt{d}_\texttt{w}$;
- using a random effect (penalized factor) for $\texttt{d}_\texttt{w}$.

In both cases the interaction between $\texttt{d}_\texttt{w}$ and other variables can be constructed as a tensor product spline. For example, model 3.2b may be written as

$$\begin{aligned}\texttt{logBS}_\texttt{d} = f_0(\texttt{d}_\texttt{w};\ 7) + f_1(\texttt{w}_\texttt{y};\ 15) + \tilde{f}_1(\texttt{d}_\texttt{w},\ \texttt{w}_\texttt{y};\ 7,\ 15) + \\ f_2(\texttt{w};\ 40) + \tilde{f}_2(\texttt{d}_\texttt{w},\ \texttt{w};\ 7,\ 40) + f_3(\texttt{w}_\texttt{y},\ \texttt{w};\ 10,\ 40),\end{aligned}$$

which is a great simplification as all "$\sum$" are gone. From now on, this tensor product spline construction will be used for modelling interaction with $\texttt{d}_\texttt{w}$ (and generally any interaction between a spline and a factor).

3.3 Motivating spatial-temporal modelling

In the previous section I have demonstrated how to build a time series model for daily logBS from any station, and in fact, as I have verified, for all 69 representative stations marked in Figure 3.21, the proposed model 3.3b proves adequate.

However, no matter how many individual daily time series are fitted and how good the prediction is at those stations, it is not possible to predict logBS away from those stations. To make such spatial or spatial-temporal prediction, we need to assume that those logBS times series are correlated, or that they vary smoothly over space hence a surface prediction is possible.

Figure 3.24 illustrated in coloured curves the estimated main effects $\hat{f}_0$, $\hat{f}_1$, $\hat{f}_2$, $\hat{f}_4$ and $\hat{f}_5$ in fitted model 3.3b for all 69 stations. It can be observed that they vary greatly between stations. However, the general pattern of each effect seems very simple, as is highlighted by the bold black curve on each graph. In spatial-temporal modelling, it is assumed that each individual time series is an observation from a spatial population. While the spatial variability between observations can be high and complicated, the mean of the population is smooth and simple.

Additive models via penalized regression splines can be used for such spatial-temporal modelling. The popolation mean can be modelled by a smooth function, like a cubic spline of week variable $\texttt{v}$ for long term trend and a thin-plate spline of spatial coordinates $\{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\}$ for spatial trend. The spatial variability across time can be modelled by a tensor product spline with a thin-plate spline margin as its spatial component and a cubic spline margin as its temporal component. In the next section, I will demonstrate this modelling approach for annual mean logBS from all stations.

3.4 Spatial-temporal modelling for yearly mean logBS

3.4.1 Description of data

The spatial-temporal dataset for yearly mean logBS, is obtained from aggregating daily logBS data per year and station. However, to ensure that this sample mean is representative for a whole year,

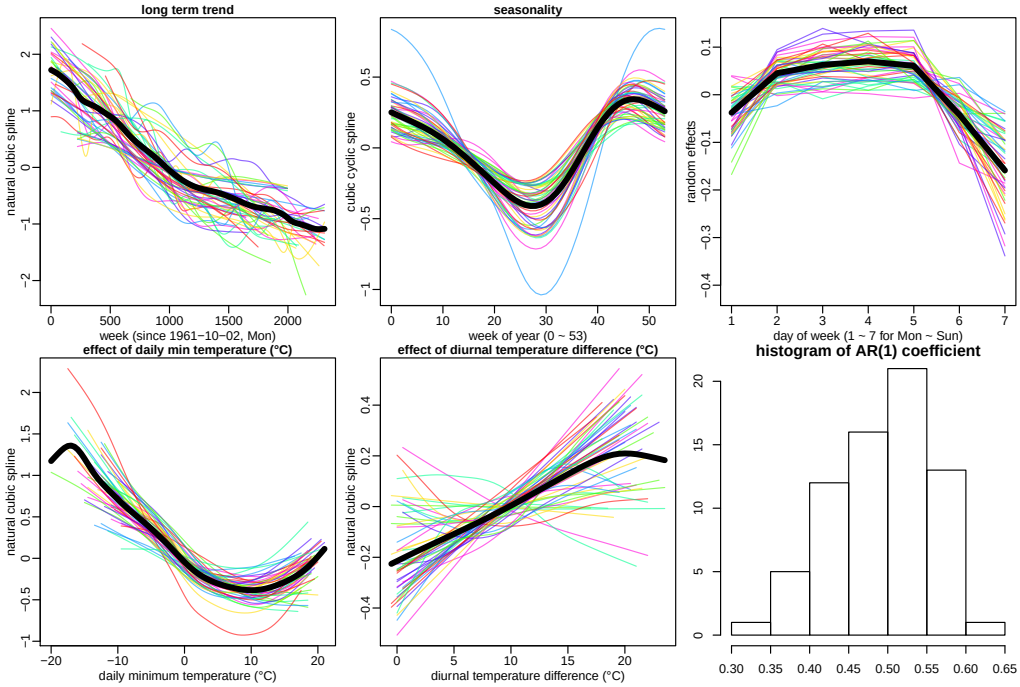


Figure 3.24: Estimated main effects (thin coloured curves) in fitted model 3.3b for all 69 stations (described in Figure 3.21) and their mean (in a bold black curve). Spatial-temporal modelling assumes each colour curve to be a sample from a population with the bold black curve as its mean. The variability between a sample and the mean is to be explained spatially: samples observed at closer spatial locations are more similar than those observed at far apart locations. The final histogram is for the estimated AR(1) coefficients in each individual time series model.

aggregation is not done if there are fewer than 273 (approximately 75% of number of days in a year) daily data. The resulting dataset has 24239 data from a total number of 2626 stations, covering a period from year 1962 to 2004. See Figure 3.25 and Figure 3.26 for some illustration of this dataset.

3.4.2 A basic model

Shaddick and Zidek (2014) attempted spatial-temporal modelling of such dataset on a study period 1966 $\sim$ 1996, concluding that there is a strong interaction between space and time (i.e., the spatial variability of logBS varies in time)[1]. A linear mixed model with the following core structure was suggested:

$$\texttt{logBS}_{\texttt{iy}} = \beta_0 + \beta_1 \texttt{y} + \beta_2 \texttt{y}^2 + \beta_{0\texttt{i}} + \beta_{1\texttt{i}} \texttt{y} + \beta_{2\texttt{i}} \texttt{y}^2 + \epsilon_{\texttt{iy}}.$$

The long term trend is modelled by a quadratic polynomial of $\texttt{y}$, whose coefficients are allowed to vary spatially. Specifically, β_0, β_1 and β_2 are fixed effect for the mean polynomial coefficients over space, and $\beta_{0\texttt{i}}$, $\beta_{1\texttt{i}}$ and $\beta_{2\texttt{i}}$ are zero-mean spatial random effects with independent Matérn covariance. Representing this model uing an additive model with splines is a type of polynomial "by" model (see model 2.2) which I have previously introduced in 2.1.1:

$$\texttt{logBS}_{\texttt{iy}} = f_1(\{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\}) + f_2(\{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\})\texttt{y} + f_3(\{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\})\texttt{y}^2 + \epsilon_{\texttt{iy}},$$

where f_1, f_2 and f_3 are thin-plate splines of $(\{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\})$, modelling spatially vary polynomial coefficients.

A quadratic polynomial is unlikely to adequately model trend at a station. For example, the trend at *Manchester 11* shown in Figure 3.15 is much more complicated that a polynomial. Furthermore, in §2.1.1 it has been explained that a polynomial "by" model suffers from Runge's phenomenon. A

[1] In fact, the long term trend plot in Figure 3.24 is also an evidence for this.

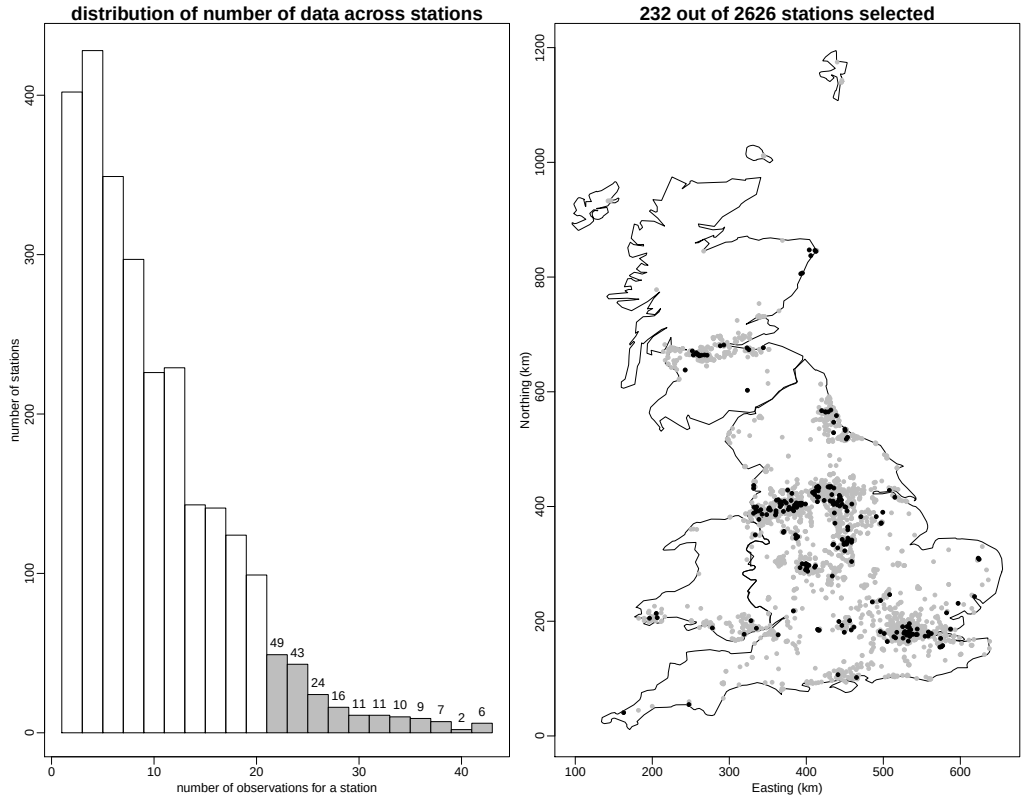


Figure 3.25: [Illustration of annual mean logBS dataset (part 1)] As we have learnt from the histogram in Figure 3.21, the majority of stations in the Network did not operate for long enough time compared with the Network itself. The histogram on the left of this Figure tells the same thing. Among 2626 stations (gray dots in the map on the right panel) in the annual logBS dataset, only 232 of them (black dots on the map) had operated for over 20 years. They contributed to 25% of all annual data.

both more flexible and numerically stable model representation is to use tensor product splines. Let us consider the following model:

Model 3.4:

$$\begin{aligned}\texttt{logBS}_{\texttt{iy}} = f_0(\texttt{E}_\texttt{i};\ 5) + f_1(\texttt{y};\ k_1) + \tilde{f}_1(\texttt{E}_\texttt{i},\ \texttt{y};\ 5,\ k_1) + \\ f_2(\{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\};\ k_2) + f_3(\texttt{y},\ \{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\};\ k_{31},\ k_{32}) + f_4(\texttt{h}_\texttt{i};\ k_4) + \epsilon_{\texttt{iy}},\end{aligned}$$

where

- f_0 is a random effect modelling the mean of logBS from stations of environmental type $\texttt{E}_\texttt{i}$;
- the general long term trend (common to all stations) is modelled by a natural cubic spline f_1 of $\texttt{y}$ with k_1 knots;
- $\tilde{f}_1$ is a tensor product spline, whose first margin is a random effect for $\texttt{E}_\texttt{i}$ and the second margin is cubic spline of $\texttt{y}$ with k_1 knots. This component models how f_1 varies for each $\texttt{E}_\texttt{i}$;
- the general spatial distribution of logBS across stations (invariant with time) is modelled by a thin-plate spline f_2 of $\{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\}$ with rank k_2 (thin-plate spline is a knots-free spline; it places knots exact all all unique locations; the k value associated with a thin-plate spline is the rank used in its low rank approximation; see §1.1.2 if you need a revision);
- the time varying spatial variability is modelled by a tensor product spline f_3, in which the first margin is a natural cubic spline of $\texttt{y}$ with k_{31} knots and the second margin is a thin-plate spline of $\{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\}$ with rank k_{32};
- f_4 is a natural cubic spline of $\texttt{h}_\texttt{i}$ with k_4 knots;
- $\epsilon_{\texttt{iy}}$ are i.i.d. model errors (the i.i.d. assumption is reasonable as the autocorrelated errors in daily logBS will be aggregated to zero over a year long span).

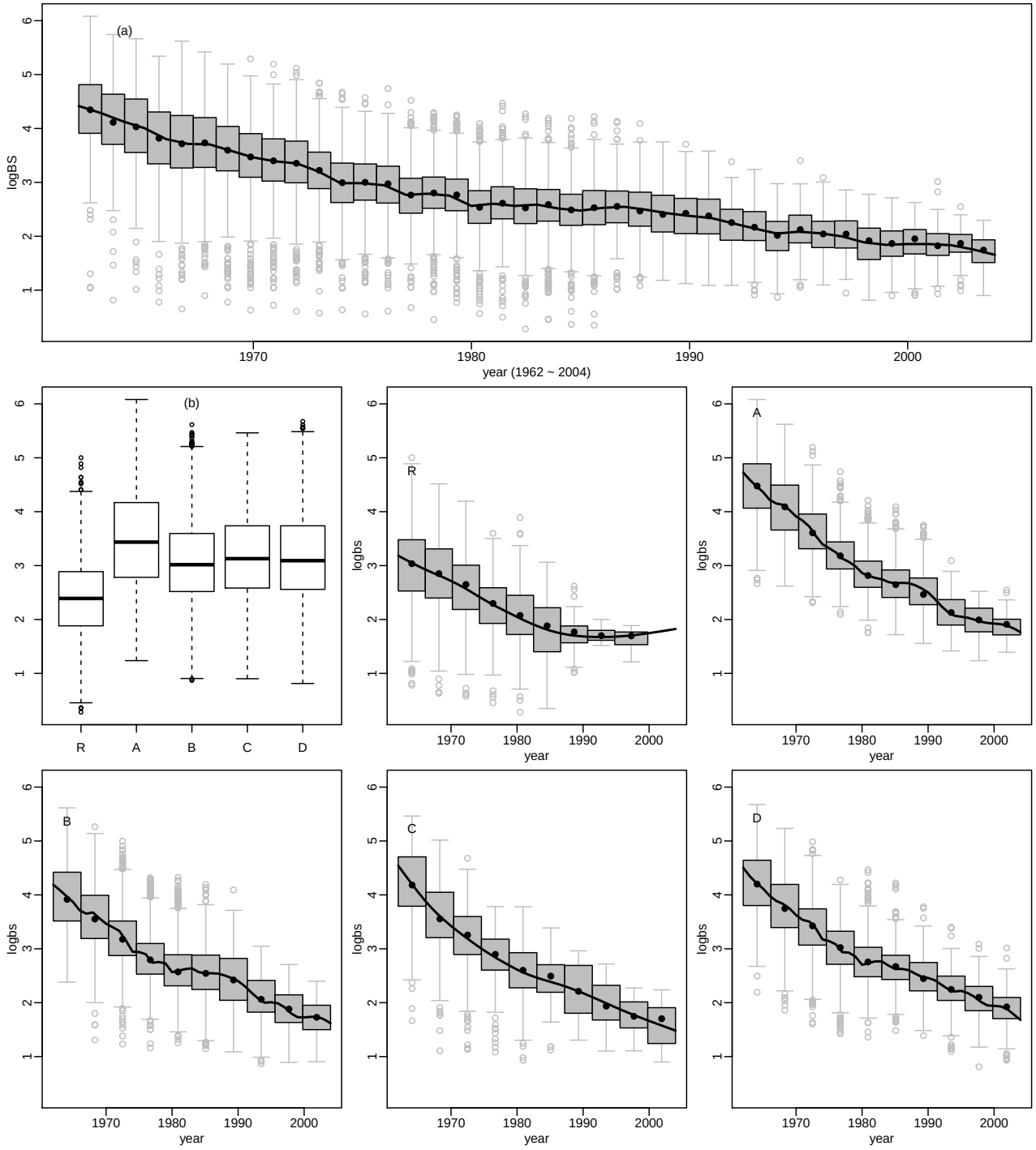


Figure 3.26: [Illustration of annual mean logBS dataset (part 2)] Panel (a) is a boxplot-alike plot (see §2.4 if you need an explanation) for logBS from all stations against year. Panel (b) is a boxplot of mean logBS over all years and all stations of a particular environmental type. It is obvious that rural areas had lower logBS. The rest of the boxplot-alike graphs sketch logBS from all stations of a particular environmental type against year. logBS declined at different rate at differernt types of areas, and the differernce between rural areas and other areas is the most notable.

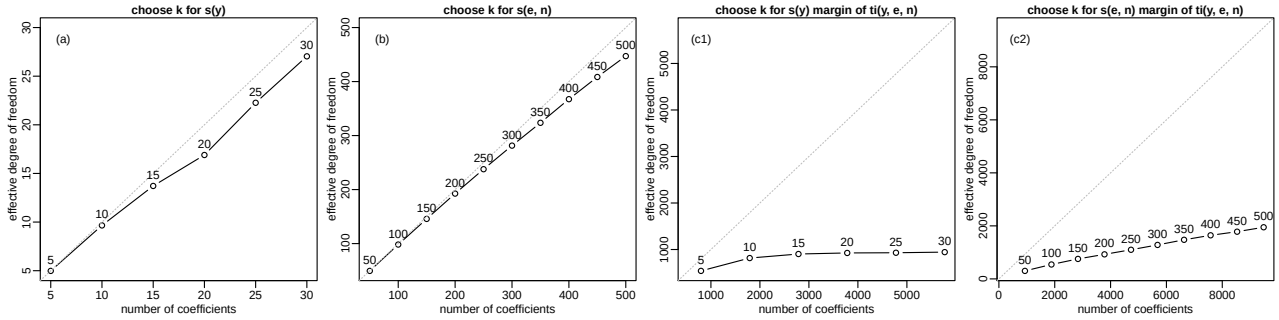


Figure 3.27: Choosing k for model 3.4 fails. The *edf* for $\hat{f}_1$ and $\hat{f}_2$ grows linearly as k increases.

Table 3.7: Refitting model 3.4 by simultaneously increasing k values for splines and check how their resulting effective degree of freedom change. Again, they grow linearly with k.

	$\hat{f}_1$	$\hat{f}_2$	$\hat{f}_3$	$\hat{f}_4$	$\tilde{f}_1$
$(k_1,\ k_2,\ \{k_{31},\ k_{32}\},\ k_4)$	*edf*				
(20, 200, {20, 200}, 10)	15.53	173.16	874.81	7.77	88.31
(25, 250, {25, 250}, 15)	21.29	211.76	1046.94	11.23	104.74
(30, 300, {30, 300}, 20)	26.16	251.78	1207.95	16.01	129.22
(35, 350, {35, 350}, 25)	31.24	287.81	1390.25	21.44	151.93
(40, 400, {40, 400}, 30)	35.86	328.67	1533.91	25.07	177.60

As usual, practial model development starts from choosing k values. To start with, consider an intial model fitting with $k_1 = k_{31} = 20$, $k_2 = k_{32} = 200$ and $k_4 = 10$. Extract the partial residuals w.r.t. $\hat{f}_1(\mathtt{y};\ 20)$ and fit those partial residuals over $f_1(\mathtt{y};\ k_1)$ for increasingly bigger k_1 and see when the effective degree of freedom plateaus. However, at this very first step some problem has occurred: the *edf* never plateaus! Maybe we can start with another model component instead? But trying $f_2(\{\mathtt{e_i},\ \mathtt{n_i}\};\ k_2)$ exposes the same issue: the *edf* grows linearly with k. Figure 3.27 demonstrates these results for $\hat{f}_1$, $\hat{f}_2$ and $\hat{f}_3$.

Apart from using partial residuals to iteratively check adequacy of k values, a more straightforward (but also more computationally expensive) way is to increase all k values, refit the model and check *edf*. Table 3.7 presents the result, and it is concerning that *edf* still grow linearly with k.

3.4.3 Including i.i.d. random effects

Observations in the last section imply that there are more variability between data across stations than what smooth functions assume: more variability between mean of logBS across stations than what $f_2(\{\mathtt{e_i},\ \mathtt{n_i}\};\ k_2)$ assumes and more variability between the shape of trend across stations than $f_1(\mathtt{y};\ k_1)$ and $f_3(\mathtt{y},\ \{\mathtt{e_i},\ \mathtt{n_i}\};\ k_{31},\ k_{32})$ assume. So when splines are parametrized with increasingly more knots to be more flexible, they continue to approximate data values producing smaller residuals. A question may be raised on whether all between-station difference should be explicitly modelled as mean via complicated regression functions; it may be a good idea to model some between-station difference as random effects and quantify their uncertainty; in this way we focus on modelling a more general and less complicated space-time relationship via spline functions.

Let us thus consider the following model

Model 3.5: model 3.4 + i.i.d. random effects

$$\begin{aligned}\mathtt{logBS_{iy}} = {} & f_0(\mathtt{E_i};\ 5) + f_1(\mathtt{y};\ k_1) + \tilde{f}_1(\mathtt{E_i},\ \mathtt{y};\ 5,\ k_1) + \\ & f_2(\{\mathtt{e_i},\ \mathtt{n_i}\};\ k_2) + f_3(\mathtt{y},\ \{\mathtt{e_i},\ \mathtt{n_i}\};\ k_{31},\ k_{32}) + f_4(\mathtt{h_i};\ k_4) + \\ & f_5(\mathtt{i};\ 2626) + f_6(\mathtt{y};\ 43) + \epsilon_{\mathtt{iy}},\end{aligned}$$

where new components compared with model 3.5 are

Table 3.8: Refitting model 3.5 by simultaneously increasing k values for splines and check how their resulting effective degree of freedom change.

	$\hat{f}_6$	$\hat{f}_5$	$\hat{f}_1$	$\hat{f}_2$	$\hat{f}_3$	$\hat{f}_4$	$\tilde{f}_1$
$(k_1,\ k_2,\ \{k_{31},\ k_{32}\},\ k_4)$	*edf*						
(20, 200, {20, 200}, 10)	35.07	2231.29	5.00	120.88	1298.83	1.00	25.94
(25, 250, {25, 250}, 15)	35.04	2194.11	5.02	140.90	1557.71	1.00	25.95
(30, 300, {30, 300}, 20)	35.07	2162.51	5.03	153.03	1868.25	1.00	25.97
(35, 350, {35, 350}, 25)	35.05	2137.41	5.04	162.86	2114.05	1.00	25.99
(40, 400, {40, 400}, 30)	35.10	2117.62	5.02	174.54	2363.10	1.00	26.00

- f_5, an i.i.d. random effect for station `i`;
- f_6, an i.i.d. random effect for year `y`.

The random effect for year should be interpreted as an adjustment to the station random effect, so that the random effect for station `i` in year `y` is $f_5(\texttt{i}) + f_6(\texttt{y})$. Estimation of random effects can be easily incorporated into standard estimation procedure of addition models with penalized regression. The design matrix of a random effect is just an matrix with dummy columns, and the associated penalty matrix is an identity matrix giving a ridge penalty. Given an estimated smoothing parameter $\hat{\lambda}_5$ for $\hat{f}_5$ for example, as well as an estimated residual variance $\hat{\sigma}^2_\epsilon$, the variance of $\hat{f}_5$ is just $\hat{\sigma}^2_\epsilon/\hat{\lambda}_5$. Predicted $\hat{f}_5$ is obtained straightforward in the solution of penalized least squares.

Because of the existence of random effects, k values can not be reliably chosen using partial residuals. There is a trade-off between the complexity in f_2 and f_5 (and likewise between f_1 and f_6) that can only be estimated jointly during model fitting. So let us take the most expensive approach: increase k values and refit the whole model. The result is presented in Table 3.8, and the following observations can be made.

- With the introduction of random effects, *edf* of $\hat{f}_1$, $\hat{f}_4$ and $\tilde{f}_1$ plateaus respectively at about 5, 1 and 26.
- The *edf* of $\hat{f}_2$ does not strictly plateau, but it grows sublinearly with k_2. For example, for *edf* : k_2 ratio is 0.6 at $k_2 = 200$, 0.56 at $k_2 = 250$, 0.51 at $k_2 = 300$, etc., which keeps dropping as k_2 increases linearly. (As a comparison, this ratio is steadily about 0.8 in Table 3.7.) In fact, slightly less than half of the coefficients are already suppressed at $k_2 = 200$, so this k value is readily sufficient.

3.4.4 Model summary and checking

$k_1 = k_{31} = 20$, $k_2 = k_{32} = 200$ and $k_4 = 10$ are sufficient for model 3.5, so let us now summarize the model and do some checking.

The effective degree of freedom for all spline functions have been given in Table 3.8. In total 6750 parameters have been used for $n = 24239$ data and the fitted model has an *edf* of 3730.7. The estimated residual variance is 0.02939 (residual sum of squares divided by (n - *edf*)); the variance of logBS data is 0.75622, so the fitted model has an adjusted R-squared at 1 - 0.02939 / 0.75622 = 96.11% which is very high. The estimated variance for station specific random effects $\hat{f}_5$ and year specific random effects $\hat{f}_6$ are respectively 0.0741 (much larger than residual variance) and 0.00272 (much smaller than residual variance).

One concern is whether the predicted random effects are really i.i.d.. If the station specific random effects turn out to exhibit spatial autocorrelation, or the year specific random effects show temporal

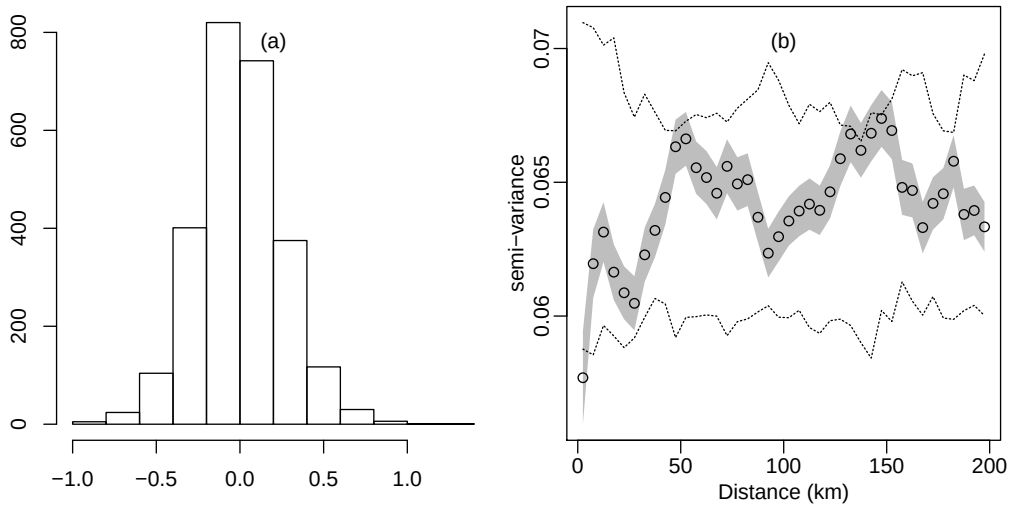


Figure 3.28: Predicted station-specific random effects $\hat{f}_5$ in fitted model 3.5. Panel (a) is their histogram, and panel (b) is their empirical variogram (computed using `geoR` package). The dots in the graph are the classic (i.e., moment) estimates for variogram, and the gray ribbon covers region within 2 standard deviation of the estimate. Envelops (dashed lines) for the estimates (by permutation of the data values on the spatial locations) are added for testing the existence of spatial autocorrelation. No violation of the i.i.d. assumption for the random effects is seen.

autocorrelation, then it is more appropriate to model them with some covariance or correlation matrix. Panel (b) of Figure 3.28 is an empirical variogram of predicted $\hat{f}_5$, and no spatial correlation is spotted. The ACF in panel (b) of Figure 3.29 confirms that there is no temporal autocorrelation between predicted $\hat{f}_6$.

Standard residual checking involves inspecting residuals against fitted values and independent variables, as is done in Figure 3.30. No violation of constant error variance assumption is seen from panel (a), and no unmodelled trend is spotted in the rest of panels. The latter is not surprising, since all these independent variables have been included in the model and sufficient k values have been chosen for all splines.

In spatial and spatial-temporal modelling it is also important to check if there are unmodelled spatial autocorrelation in residuals. Figure 3.31 produces empirical variograms for spatial residuals every three years. It seems that the thin-plate spline $f_2(\{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\};\ k_2)$ and the tensor product spline $f_3(\texttt{y},\ \{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\};\ k_{31},\ k_{32})$ have adequately modelled all spatial-temporal variability.

3.4.5 Visualization of model prediction

So far the fitted model 3.5 looks reasonable. Now let us visualize its components and demonstrate how it makes prediction.

Figure 3.32 illustrates various main effects in the fitted 3.5.

- Panel (a) shows the estimated mean long term trend $\hat{f}_1$. Its complexity appears to be somewhere between a linear line and a quadratic polynomial. Or it looks like two linear line segments that join at around year 1980. Before 1980 the lines descends fast, and slower afterwards.
- Panel (b) is the estimated linear line for elevation effect $\hat{f}_4$. That the confidence interval shrinks to zero somewhere at the middle may be confusing, but it is expected since in additive models all additive components are subject to centring constraints for better identifiability (and that is why all splines in the Figure cross zero). A linear line has only 1 degree of freedom, thus has no degree of freedom at centring points.

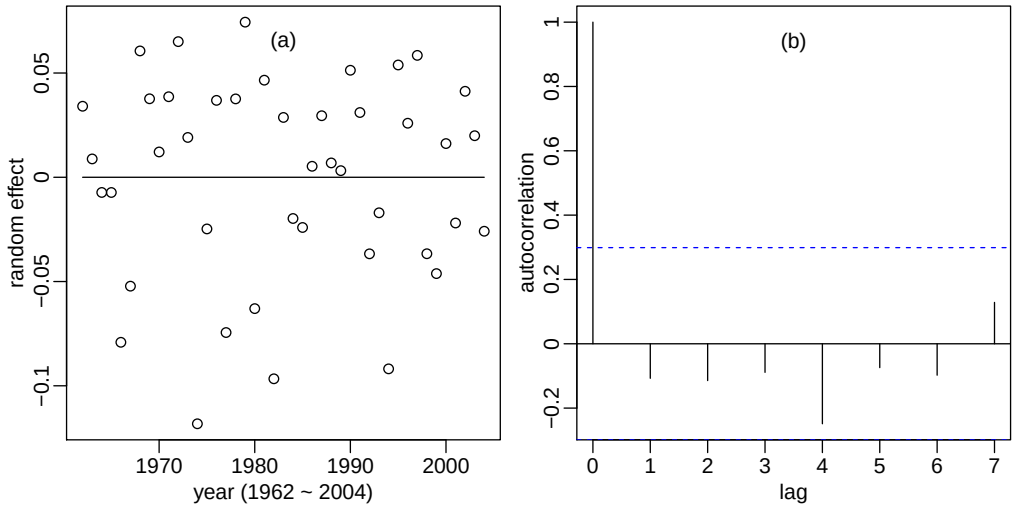


Figure 3.29: Predicted year-specific random effects $\hat{f}_6$ in fitted model 3.5. Panel (a) is a scatter plot of $\hat{f}_6$ against year. A smoothing spline (black line) is fitted to the data but ends up with a horizontal line at 0, confirming that there is no trend in $\hat{f}_6$. Panel (b) is an autocorrelation function (ACF) for $\hat{f}_6$. Although these random effects are ordered in time, no autocorrelation is spotted so the i.i.d. assumption is validated.

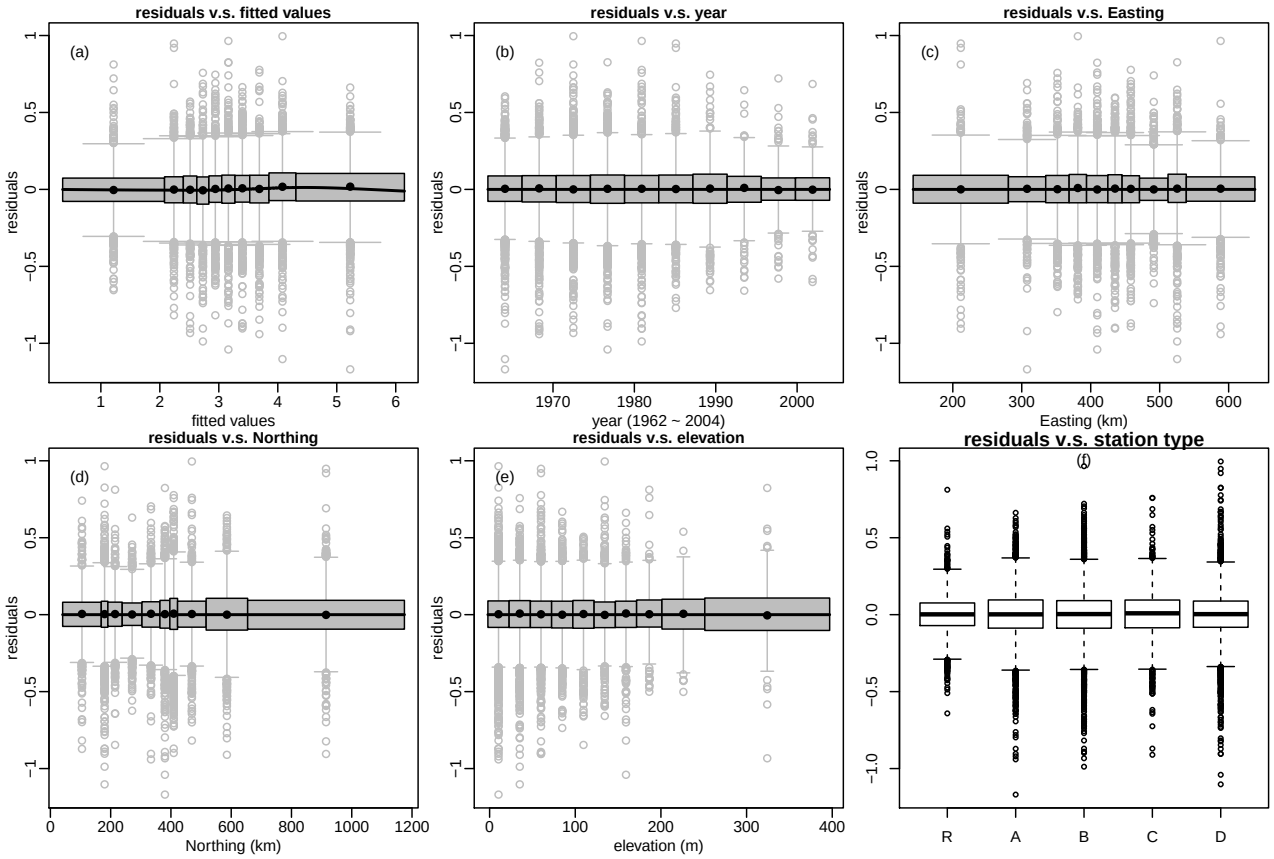


Figure 3.30: Inspecting residuals of fitted model 3.5 against fitted values (panel (a)) and other covariates ((b) to (e)). No violation of constant error variance assumption is seen from panel (a), and no unmodelled trend is spotted in the rest of panels. The latter is not surprising, since all these independent variables have been included in the model and sufficient k values have been chosen for spline functions. See §2.4 if you need an explanation for these boxplot-alike graphs.

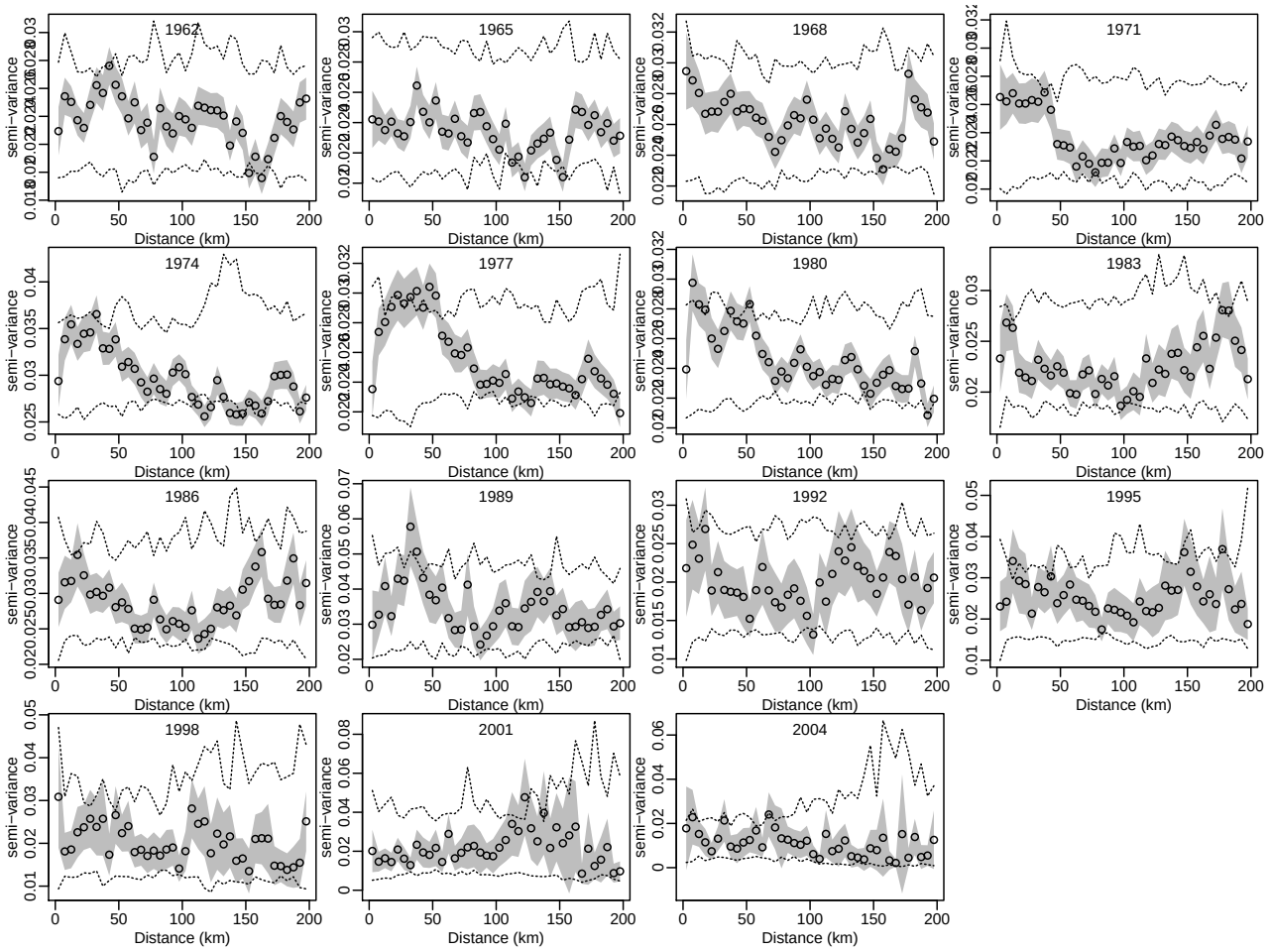


Figure 3.31: Residual checking of fitted model 3.5 in spatial domain every three years, i.e., at 1962, 1965, ..., 2004. No unmodelled spatial autocorrelation is seen.

- Panel (c) is the estimated mean logBS at different environmental areas. "Rural" ("R") is the reference level for this factor variable hence is contrasted, i.e., it has zero mean and zero standard error. Estimates at other levels are in fact the mean difference from "R". It is clear that all other areas have higher mean than rural areas. This graph verifies what was previously observed from data (see the boxplot in panel (b) of Figure 3.26).
- Panel (d) is the estimated mean long term trend $\hat{f}_1 + \tilde{f}_1$ for annual logBS at different types of areas / stations. I choose to illustrate this quantity because it is more interpretable than $\tilde{f}_1$ alone that describes the mean trend difference between a level and the reference level "R". It is clear that logBS declined at a different rate in rural areas; the decline rate for other four levels are, while different, more similar. This estimated result agrees with what was observed from data (see Figure 3.26).
- Panel (e) is the estimated thin-plate spline $\hat{f}_2$ for time invariant mean logBS over space. A contour plot is used for surface visualization as it is more informative / readable than a perspective plot. Pixels are coloured for further enhancement. The colouring scheme used is (roughly) green, yellow, orange, red, purple from low logBS to high logBS values. It can be seen that central England and Northeastern England are areas with high Smoke concentration.

Figure 3.33 illustrates how the fitted model 3.5 predicts logBS when its model terms are added together. 4 stations with different number of operation years are taken as examples. See Figure caption for more details.

The visualization of tensor product spline $\hat{f}_3$ is much more difficult as it is a 4D surface. Treating the 2D spatial coordinates as a single dimension allows us to inspect the transects line along the time dimension, that is, how the long term trend at a specific spatial location differs from the mean trend

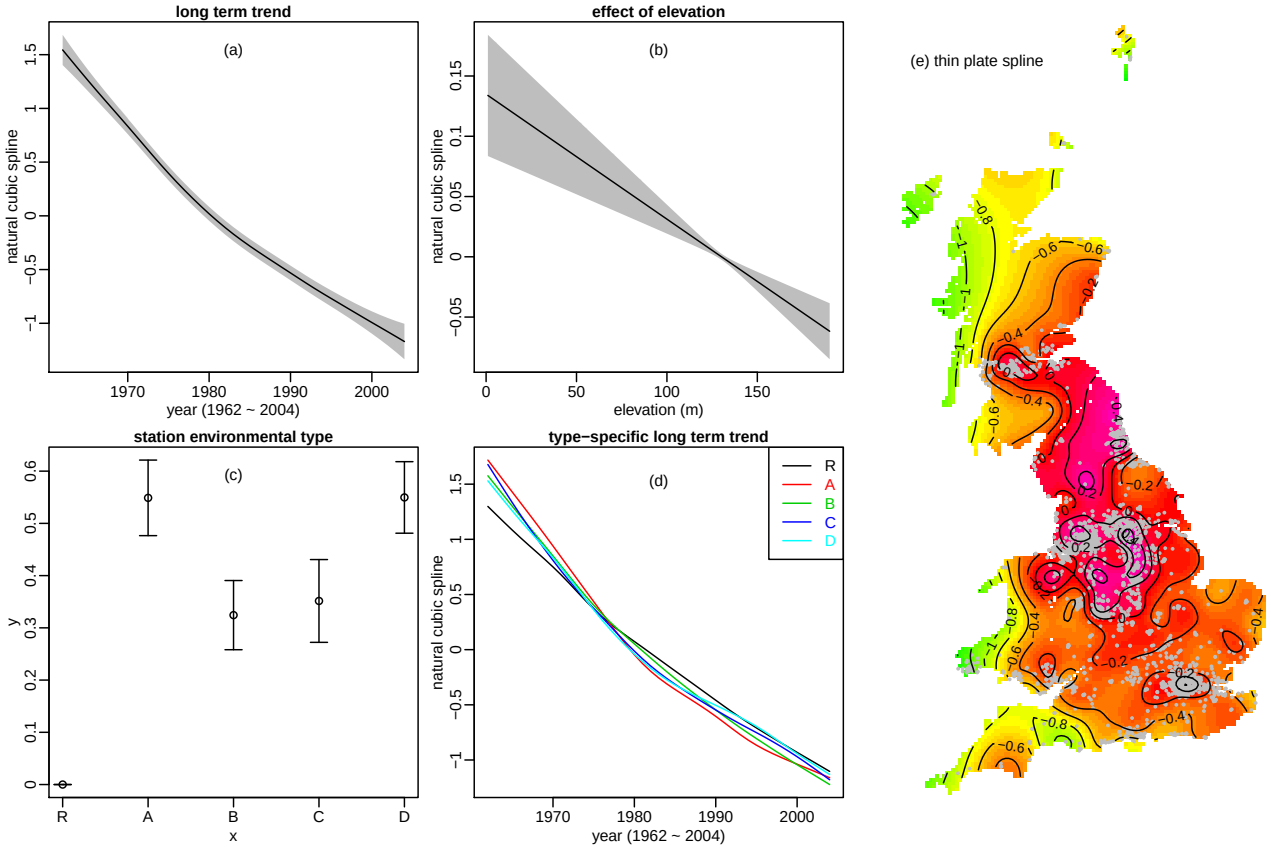


Figure 3.32: Estimated main effects in fitted model 3.5. Panel (a) is the estimated mean long term trend $\hat{f}_1$ for annual logBS (the gray ribbon covers regions within 2 standard error from the estimate); panel (b) is the estimated spline / line $\hat{f}_4$ for elevation; panel (c) is the estimated mean logBS $\hat{f}_0$ at different types of areas / stations (the whiskers stretch to 2 standard error from the estimate; note that "rural" type is the reference level and is contrasted, so it has mean 0 and standard error 0); panel (d) is the estimated mean long term trend $\hat{f}_1 + \tilde{f}_1$ for annual logBS at different types of areas / stations; panel (e) is the estimated thin-plate spline $\hat{f}_2$ for time invariant mean logBS over space. See main text for more detailed explanations.

$\hat{f}_1$. Given the simple smooth function in Figure 3.32, the high variability in data (see Figure 3.26) but high R-squared of the fitted model, one can expect this tensor product spline to be complicated. Figure 3.34 sketches transects of the spline through all 2626 stations. Generally those transects are multimodal "wave" like curves so that the mean trend $\hat{f}_1$ can be "twisted" when added by them. However, the four evident fast ascending lines are very concerning, as they imply that logBS will be predicted to soar at associated stations. This is very likely due to the extrapolation of splines when for example, a station was closed at early years but prediction is asked for the entire 43 years. With a bit of efforts, it can be identified that those four lines are for the four stations in Shetland which are far apart from the majority stations in the Network. (Referring to the map in Figure 3.25, they are on the northernmost small island.) Figure 3.35 demonstrates the extrapolation at those stations.

It is then natural to consider removing those stations from Shetland when fitting model, however, the extrapolation problem exists with other stations, too. Some more sophisticated solution is needed to deal with this problem. For the moment, I will set aside this issue.

The aim of spatial-temporal modelling is to make prediction away from sampling locations, or simply put, to produce high-resolution mapping of logBS over space and time. Figure 3.36 illustrates this, by predicting logBS on a 5km $\times$ 5km grid over Great Britain every 8 years. The general decline of logBS can be clearly seen. Note that the extrapolation problem for Shetland can also be spotted.

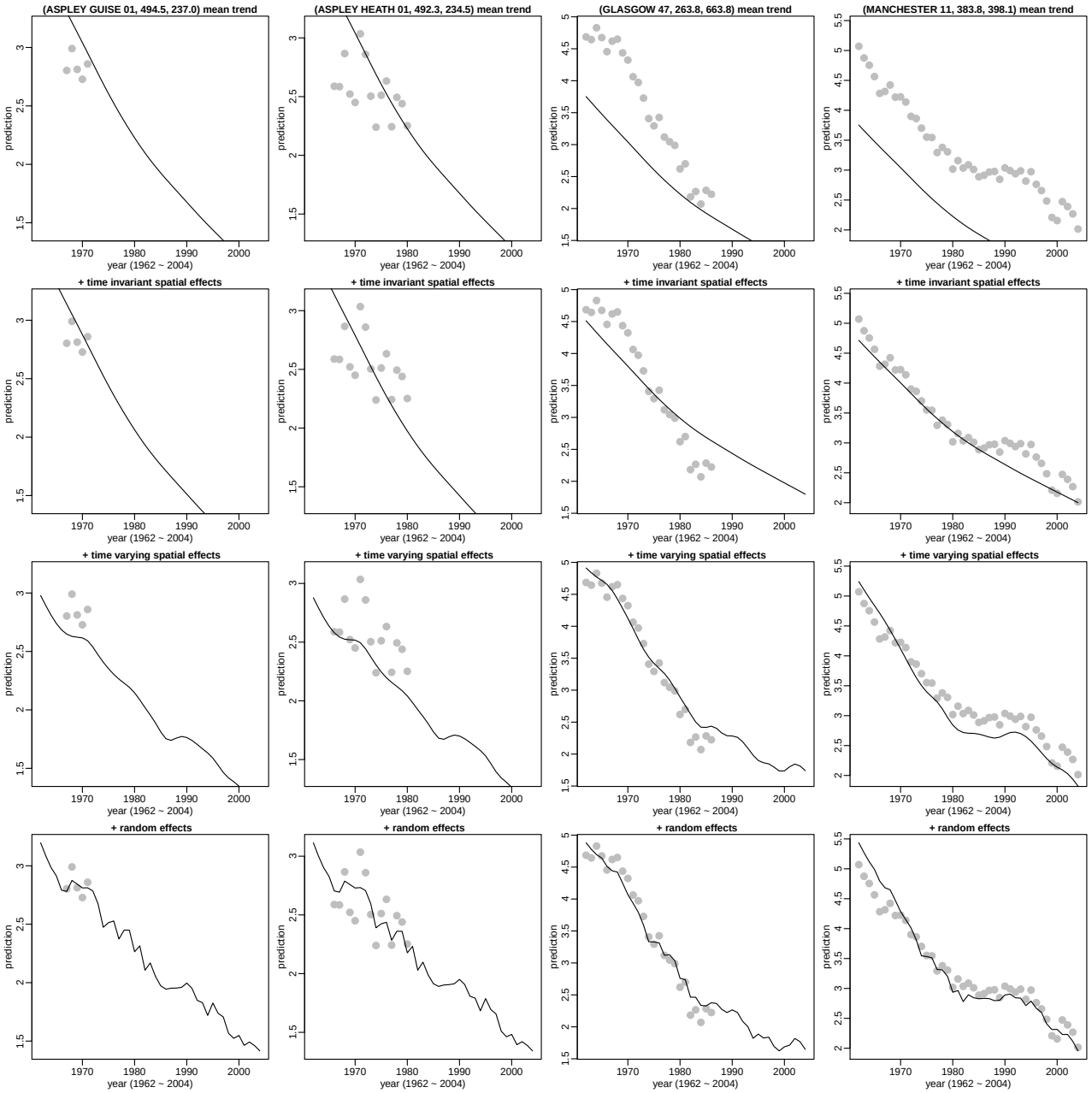


Figure 3.33: An illustration of how the fitted model 3.5 predicts logBS when its model terms are added together. Four stations with different length of monitoring history are chosen. The first (leftmost) column panel is for station *ASPLEY GUISE 01* at {494.5 (Easting), 237.0 (Northing)}; it operated for 5 years. The second column panel is for station *ASPLEY HEATH 01* at {492.3, 234.5}, which operated for 15 years. Note that this station is located very close to the first one. The third column panel is for station *GLASGOW 47* at {263.8, 663.8}, with 25 years' service. The final column panel is for the *Manchester 11* that has been thoroughly studied in a previous section. For column panels, from top rows to bottom rows, these model components are added: 1) Mean long term trend $\hat{f}_1$ (model intercept included). This component is identical for all stations, although the curves in the graphs look different due to different scale of the vertical axis; 2) Time invariant spatial effects $\hat{f}_2 + \hat{f}_4 + \hat{f}_0$; This component yields a vertical shift to the previous component; 3) time varying spatial effects $\hat{f}_3 + \tilde{f}_1$; This component "twists" the curve so that it has a closer shape to that of data. 4) predicted random effects $\hat{f}_5 + \hat{f}_6$. This component adds another vertical shift as well as fine-scaled adjustment (that is not easily modelled by a smooth function) to the predicted curve. As more terms are added, the predicted curve approximates data better. Note that the final curves for the first two stations look extremely similar. This is because the stations are very closely located so there is little spatial variability in prediction.

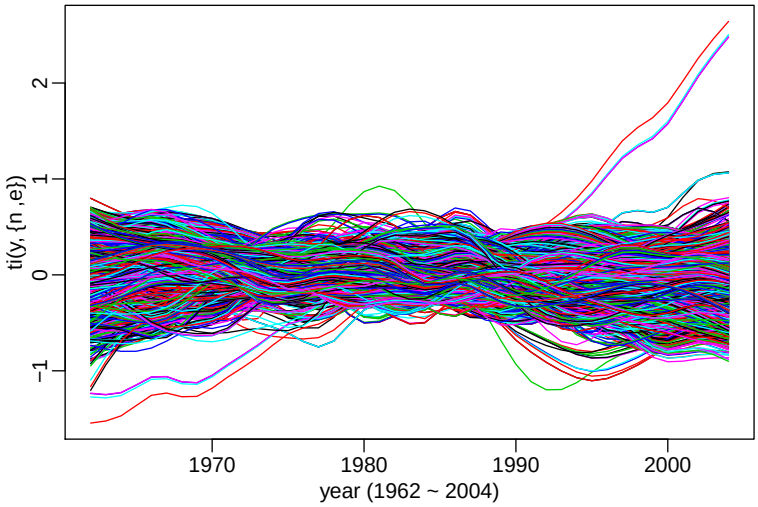


Figure 3.34: Transects of tensor product spline $\hat{f}_3$ at all stations. Most of the curves are multimodal "wave" like curves which only lead to mild "twist" of $\hat{f}_1$, but there are four evident fast ascending lines associated with four stations at Shetland that would cause unreasonable extrapolation when predicting logBS. See Figure 3.35.

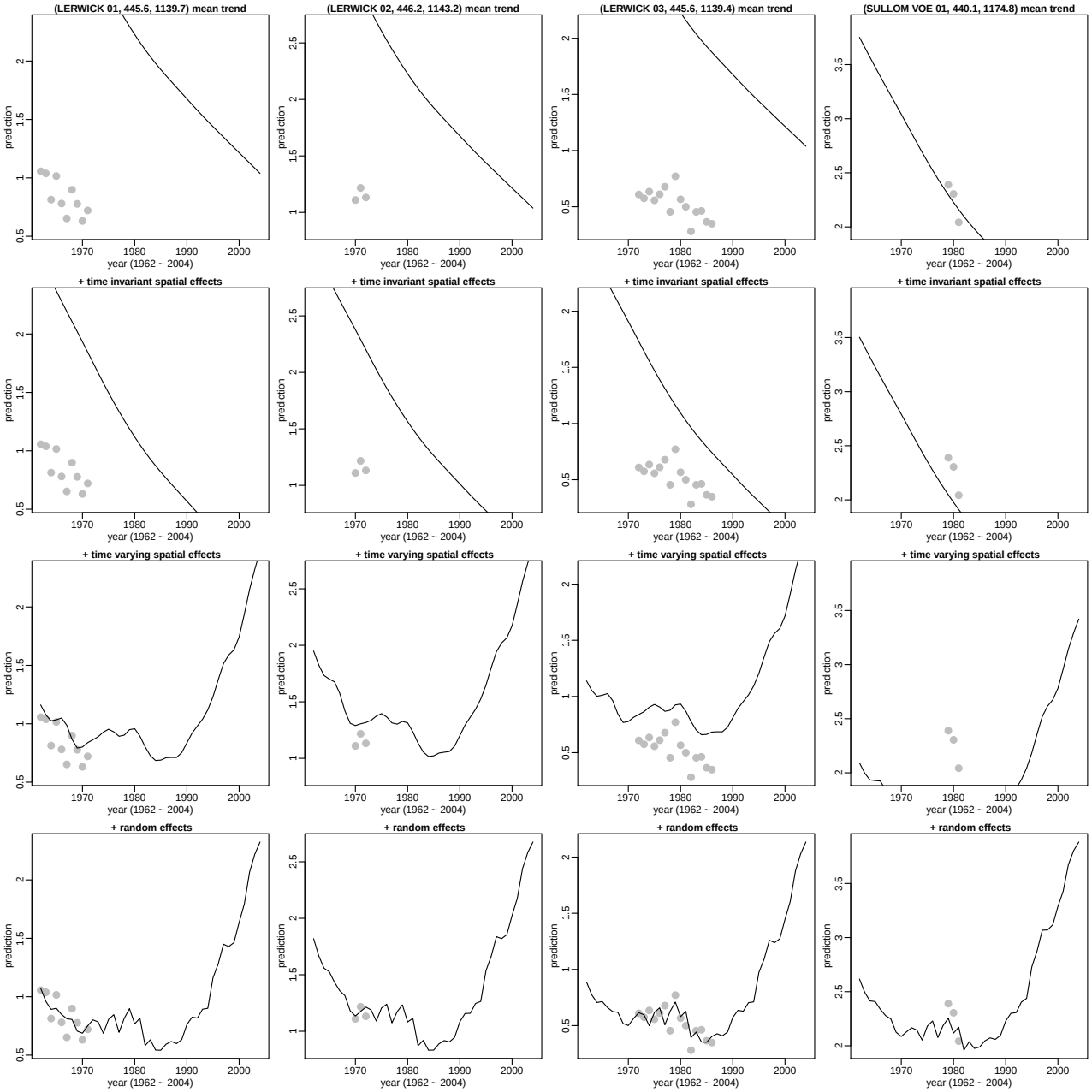


Figure 3.35: The extrapolation problem at Shetland, Scotland.

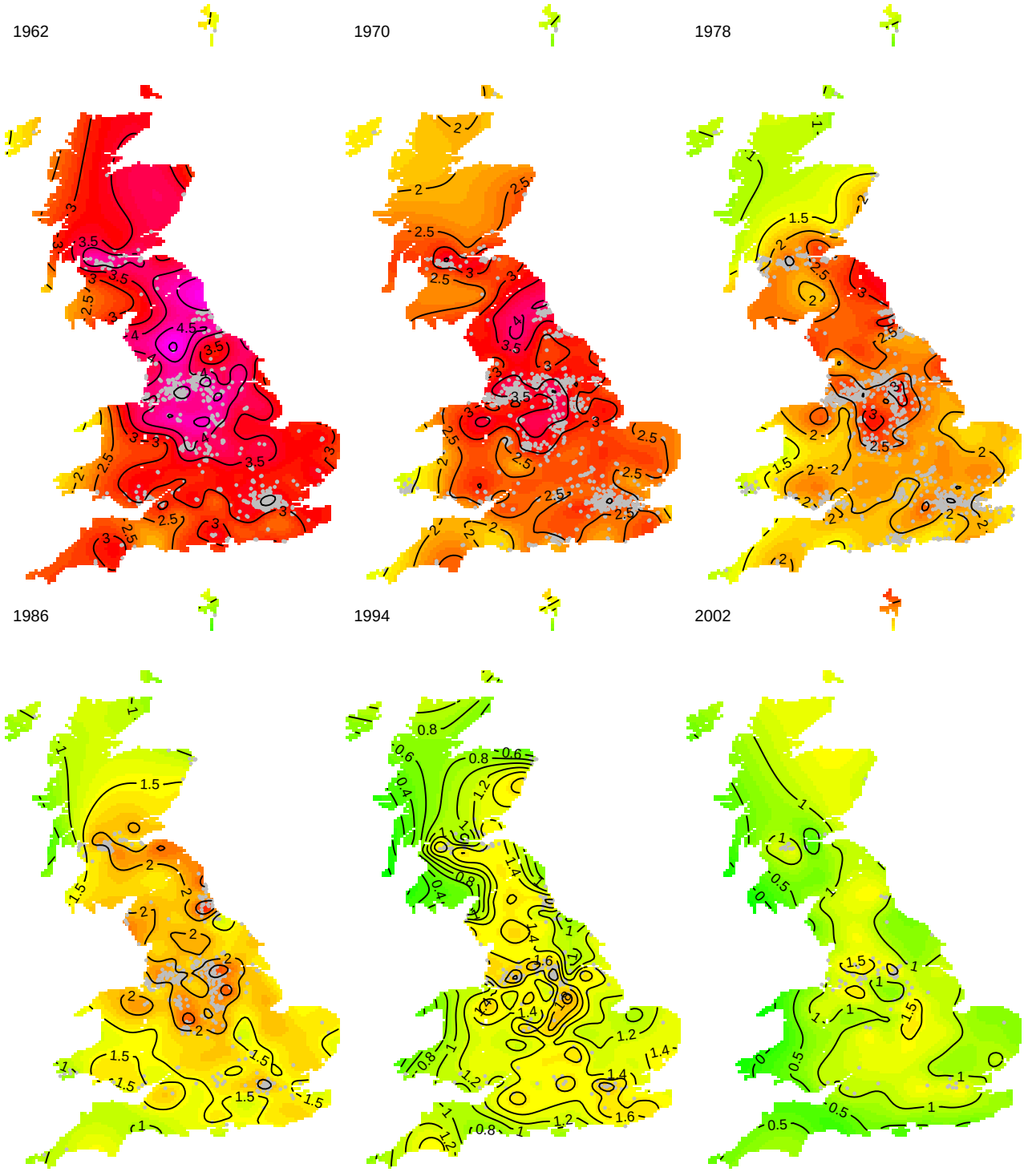


Figure 3.36: Spatial-temporal prediction of logBS on a 5km × 5km spatial grid over Great Britain every 8 years using fitted model 3.5.

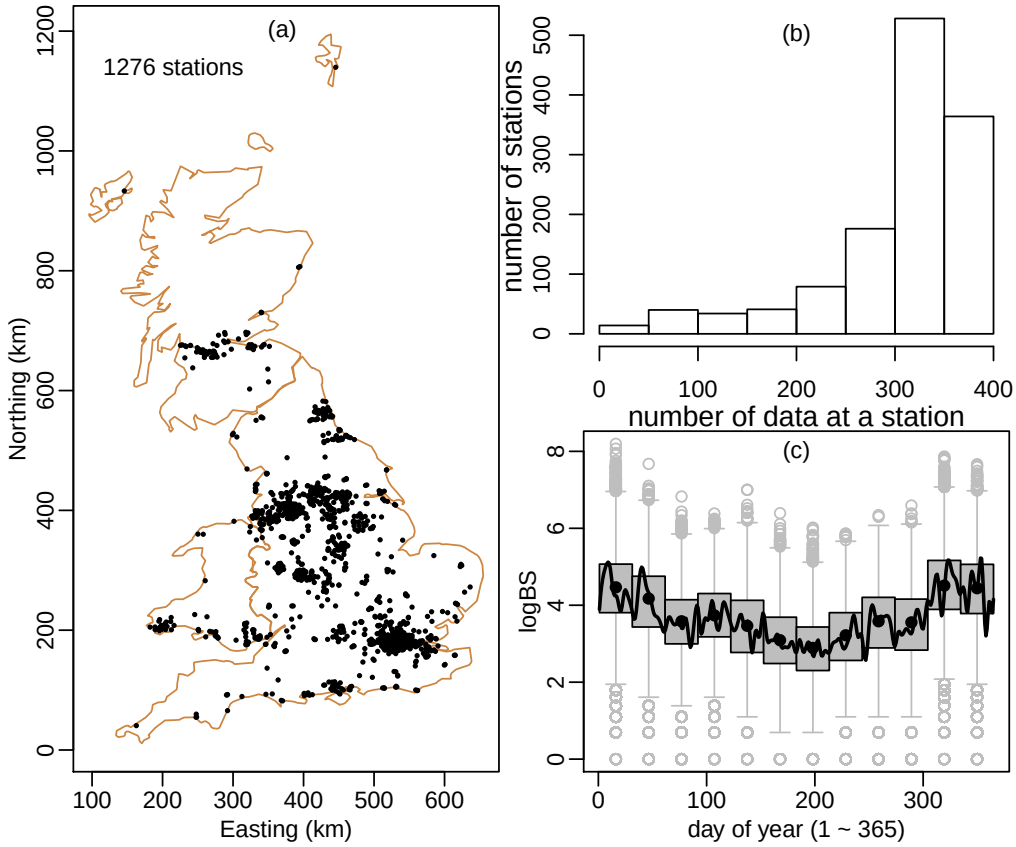


Figure 3.37: [Illustration of daily logBS dataset in year 1967 (part 1)] Panel (a) shows all 1276 operating stations in year 1967. Panel (b) is a histogram for the number of stations with a certain number of daily data. The majority of stations had operated for over 300 days. Panel (c) is a boxplot-alike graph for logBS from all stations against day of year.

3.4.6 Summary

In this section the annual mean logBS dataset is used to demonstrate how additive models with penalized regression splines can be used for spatial-temporal modelling. The following message can be learned.

- Spline functions alone can not adequately model all spatial or temporal variability in logBS data. I.i.d. random effects on both space and time domain are needed to model non-smooth change if logBS over space and time.
- The extrapolation problem is inherent for splines. Some furthe investigation on this issue is required in future.

3.5 Spatial-temporal modelling for daily logBS in 1967

3.5.1 Description of data

In this section, I will attempt developing a model for daily logBS in year 1967, when the size of the Network was at its peak with 1276 sites. The dataset comprises 387253 daily data, approximately 4.1% of the complete daily logBS dataset. See Figure 3.37 and Figure 3.38 for some visualization of this dataset.

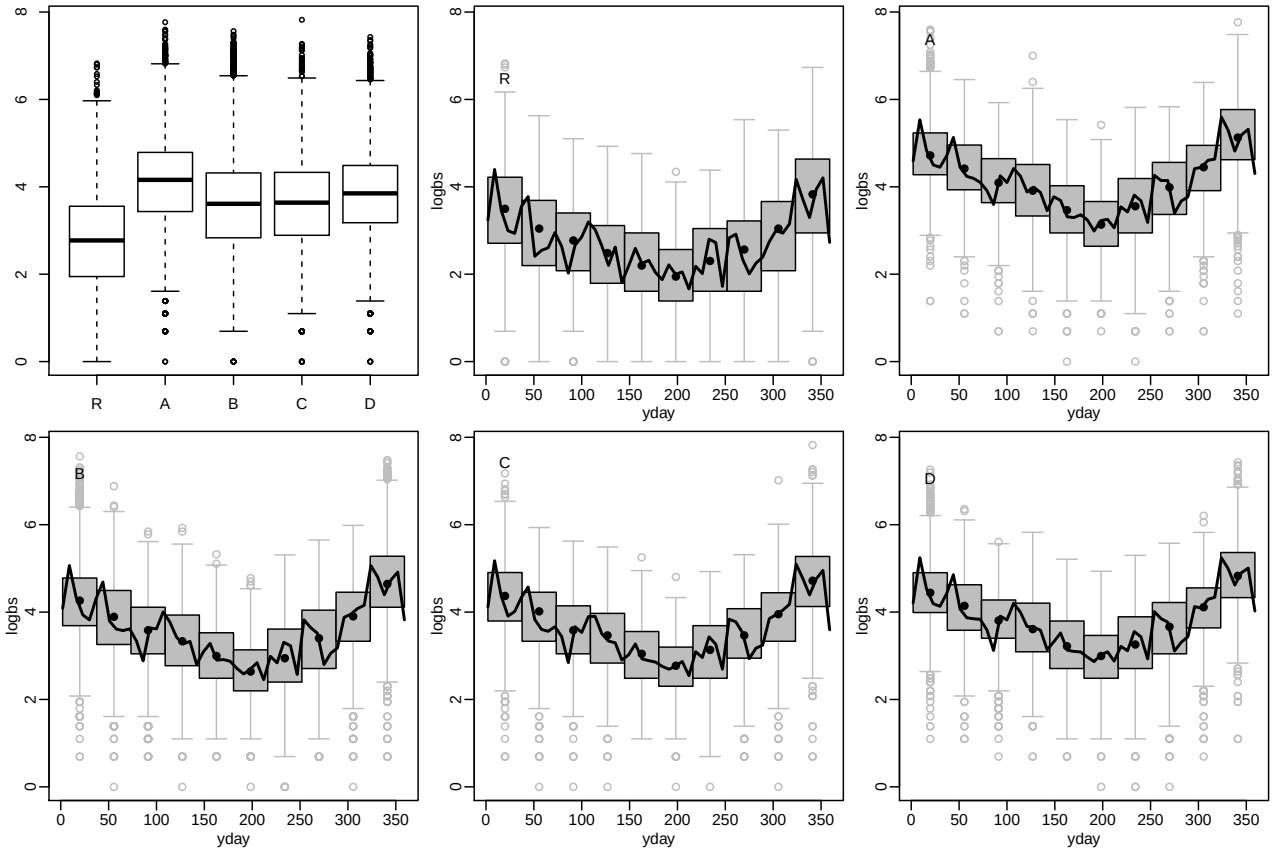


Figure 3.38: [Illustration of daily logBS dataset in year 1967 (part 3)] The first boxplot is for the mean logBS over all days and all stations of a particular environmental type. It is clear that logBS differs in mean at different environmental areas. The subsequent boxplot-alike graphs sketch logBS against day of year for all stations of the same environmental type. There does not seem to be any difference between the mean seasonality of logBS at different environmental areas.

3.5.2 Building a model for Monday data

Since daily data has strong autocorrelation, I will thin the data and first investigate Monday data. This subset of data has 54386 daily data from 1275 stations (1 fewer than the total number stations in the year).

In fact, the modelling experience established in previous sections has provided rich information to help building a model here.

- From Figure 3.24 in §3.3, it can be learned that seasonality of logBS and temporature effects on logBS vary spatially, so tensor product splines are needed to model interaction between spatial coordinates $\{\mathtt{e_i},\ \mathtt{n_i}\}$ and week of year $\mathtt{w_y}$, as well as $\{\mathtt{e_i},\ \mathtt{n_i}\}$ and temperatures $\mathtt{T^0_{id}}$, $\mathtt{T^*_{id}}$.
- From §3.4.6, it can be learned that station-specific random effects and week-specific random effects are necessary to model non-smooth change in spatial and temporal variability that is difficult to model by spline functions.

In addition,

- Figure 3.38 shows no noticeable difference in (the shape of) seasonality of logBS between different environmental areas, so there is no need to model interaction between week of year $\mathtt{w_y}$ and environmental types $\mathtt{E_i}$.

- The number of knots of natural cubic splines or the rank of thin-plates that are adequate for modelling the effects of various variables are also known.

So it is straightforward to propose the following model as a start:

Model 3.6: A model for Monday logBS

$$\begin{aligned}\texttt{logBS}_{\texttt{iw}} = {} & f_0(\texttt{E}_\texttt{i}) + f_1(\texttt{w}_\texttt{y};\ 20) + f_2(\{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\};\ 200) + f_3(\texttt{w}_\texttt{y},\ \{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\};\ 20,\ 200) + \\ & f_4(\texttt{T}^0_\texttt{id};\ 15) + f_5(\texttt{T}^0_\texttt{id},\ \{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\};\ 15,\ 200) + \\ & f_6(\texttt{T}^*_\texttt{id};\ 15) + f_7(\texttt{T}^*_\texttt{id},\ \{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\};\ 15,\ 200) + \\ & f_8(\texttt{T}^0_\texttt{id},\ \texttt{T}^*_\texttt{id};\ 15,\ 15) + f_9(\texttt{i};\ 1275) + f_{10}(\texttt{w}_\texttt{y};\ 52) + f_{11}(\texttt{h}_\texttt{i};\ 10) + \epsilon_{\texttt{iw}},\end{aligned}$$

where

- $f_0(\texttt{E}_\texttt{i};\ 5)$ is a random effect modelling the mean of logBS from stations of environmental type $\texttt{E}_\texttt{i}$;
- $f_1(\texttt{w}_\texttt{y};\ 20)$ is a cubic cyclic spline with 20 knots for $\texttt{w}_\texttt{y}$, modelling the mean seasonality of logBS at all stations;
- $f_2(\{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\};\ 200)$ is a rank-200 thin-plate spline for spatial coordinates of stations, modelling the mean logBS over space;
- f_3 is a tensor product spline modelling how station-specific seasonality deviates from f_1; its first margin is a cubic cyclic spline with 20 knots for $\texttt{w}_\texttt{y}$ and its second margin is a rank-200 thin-plate spline for spatial coordinates of stations;
- $f_4(\texttt{T}^0_\texttt{id};\ 15)$ is a natural cubic spline with 15 knots for daily minimum temperature $\texttt{T}^0_\texttt{id}$, modelling the mean effect of this variable at all stations;
- f_5 is a tensor product spline modelling how station-specific $\texttt{T}^0_\texttt{id}$ effect deviates from f_4; its first margin is a natural cubic spline with 15 knots for $\texttt{T}^0_\texttt{id}$ and its second margin is a rank-200 thin-plate spline for spatial coordinates of stations;
- $f_6(\texttt{T}^*_\texttt{id};\ 15)$ is a natural cubic spline with 15 knots for diurnal temperature difference $\texttt{T}^*_\texttt{id}$, modelling the mean effect of this variable at all stations;
- f_7 is a tensor product spline modelling how station-specific $\texttt{T}^*_\texttt{id}$ effect deviates from f_6; its first margin is a natural cubic spline with 15 knots for $\texttt{T}^*_\texttt{id}$ and its second margin is a rank-200 thin-plate spline for spatial coordinates of stations;
- $f_8(\texttt{T}^0_\texttt{id},\ \texttt{T}^*_\texttt{id};\ 15,\ 15)$ is a tensor product spline modelling the interaction between two temperature variables;
- $f_9(\texttt{i};\ 1275)$ is an i.i.d. random effect for 1275 stations;
- $f_{10}(\texttt{w}_\texttt{y};\ 52)$ is an i.i.d. random effect for week of year (there are 52 Mondays in this year);
- $f_{11}(\texttt{h};\ 10)$ is a cubic spline for elevation $\texttt{h}_\texttt{i}$;
- $\epsilon_{\texttt{iw}}$ is an i.i.d. model error.

3.5.3 Model summary and checking

Table 3.9 summarizes the effective degree of freedom for all splines in fitted model 3.6. They are much smaller than the specified k values so k values are adequate. This model involves 10932 parameters with an *edf* of 3928.32 for 54386 data. The estimated residual variance is 0.1954, and the sample

Table 3.9: effective degree of freedom for all splines in fitted model 3.6

$\hat{f}_0$	$\hat{f}_1$	$\hat{f}_2$	$\hat{f}_3$	$\hat{f}_4$	$\hat{f}_5$	$\hat{f}_6$	$\hat{f}_7$	$\hat{f}_8$	$\hat{f}_9$	$\hat{f}_{10}$	$\hat{f}_{11}$
3.98	4.74	106.16	1419.05	4.81	624.35	4.50	520.67	88.27	1103.86	45.97	1.00

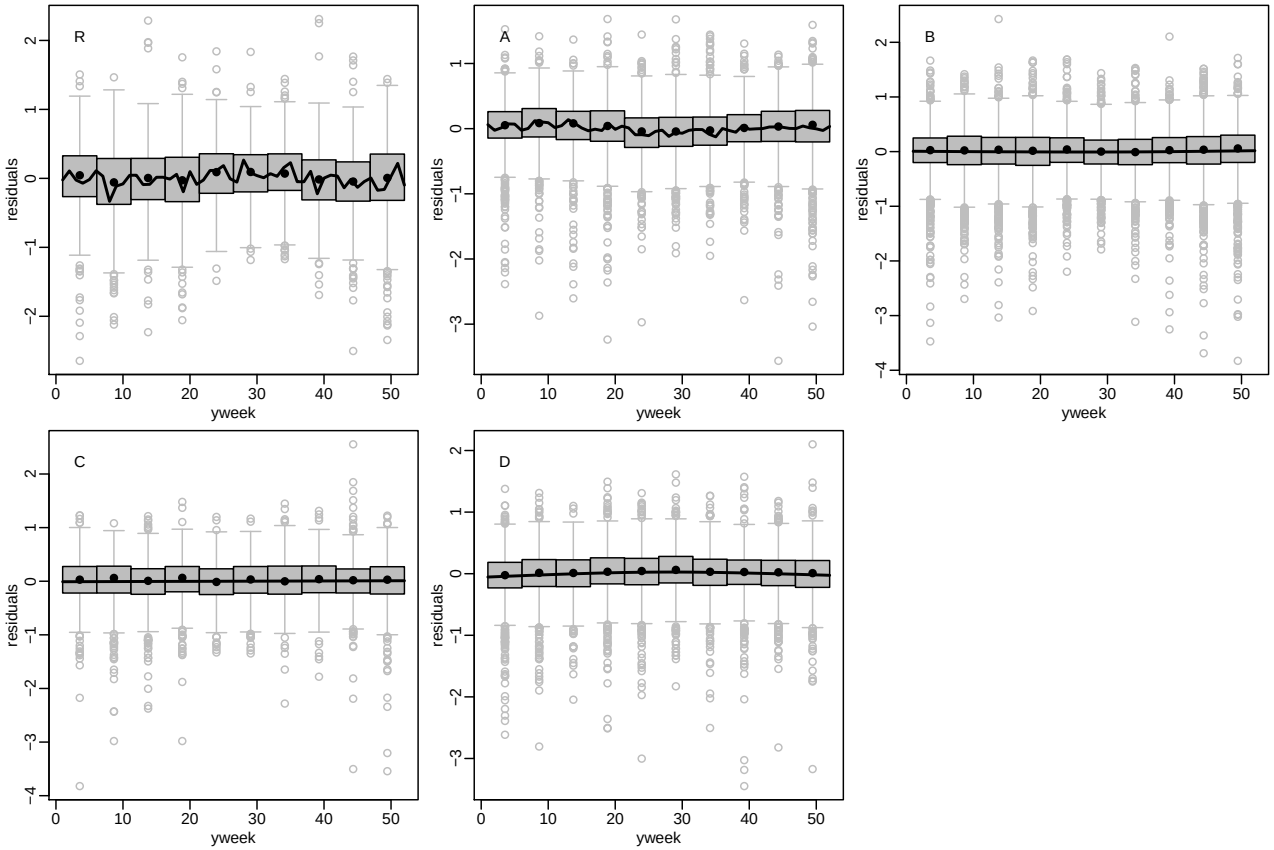


Figure 3.39: Inspecting residuals of fitted model 3.6 from each environment area against week of year. For type "B", "C" and "D", there is strictly no unmodelled trend; for type "R" and "A", the smoothing spline has fitted some trend, but it looks more like random noise. So there is no need to include an interaction between type and week of year in the model.

variance for logBS data is 1.2642, so the fitted model has an adjusted R-squared of 84.54%. Estimated variance for weekly random effects and station-specific random effects are respectively 0.0892 and 0.1116.

As I commented earlier on Figure 3.38, there does not seem to be any difference between seasonality of logBS from different environment areas. However, it will still be good to confirm this from residuals of fitted model 3.6. Figure 3.39 produces boxplot-alike graphs for residuals from different environment areas against week of year. For type "B", "C" and "D", there is strictly no unmodelled trend; for type "R" and "A", the smoothing spline has fitted some trend, but it looks more like random noise. So I believe there is no need to include an interaction between type and week of year in the model.

Figure 3.40 does some other checking on residuals. The "residuals v.s. fitted" plot assures that residuals have constant variance independent of regression mean. There is no need to plot residuals against variables included in the model; as is previously demonstrated in Figure 3.30, there will be no unmodelled trend w.r.t. those variables when k values are sufficient. It is more interesting to plot residuals against rainfall to see if rainfall can potentially improve the model. However, from panel (b) it appears that the effect of rainfall is negligible. Only rainfall over 300 mm per month seems to reduce logBS slightly, however, there are not many rainfall records above this threshold, so such conclusion is not strong. Once again, the effects of rainfall have to be investigated when there are more data. Panel (c) is a boxplot for autocorrelation function (ACF) of residuals from every stations. On average, there is no sign of residual autocorrelation. This is somehow expected, since I have thinned daily data only retaining those from Monday for model development.

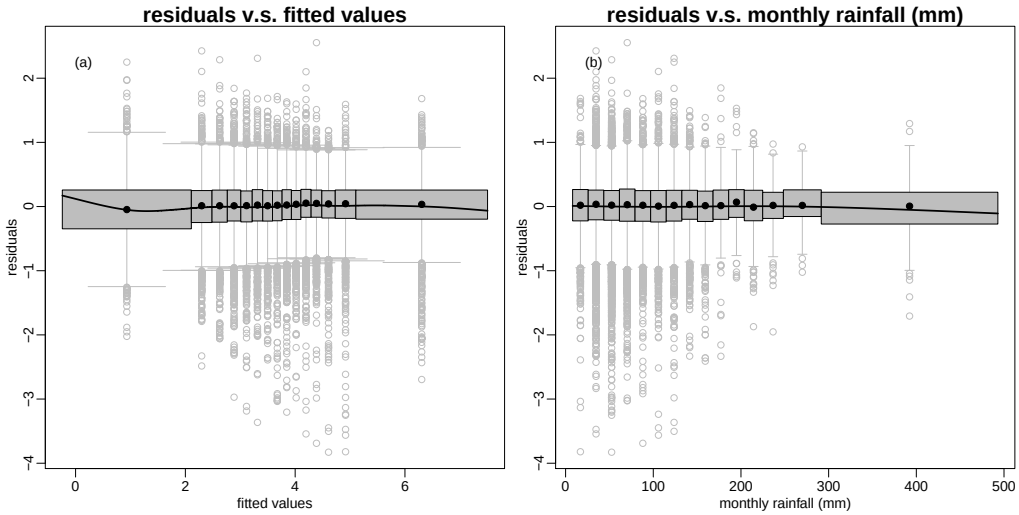


Figure 3.40: Some residual checking for fitted model 3.6. Panel (a) inspects residuals against fitted values and residuals appear to have a constant variance that is independent of regression mean. Panel (b) plots residuals against rainfall, a variable that is not yet in the model. It appears that the effect of rainfall is negligible. Only rainfall over 300 mm per month seems to reduce logBS slightly, however, there are not many rainfall records above this threshold, so such conclusion is not strong. Panel (c) is a boxplot for autocorrelation function (ACF) of residuals from every stations. On average, there is no sign of residual autocorrelation so the i.i.d. model error assumption is validated.

Random effects must also be checked to validate their i.i.d. assumption. Figure 3.41 includes a histogram and ACF for predicted weekly random effects. No correlation is seen at all lags great than 0. Figure 3.42 includes a histogram and empirical variogram for predicted station-specific random effects. There is no sign of spatial autocorrelation.

Now let us visualize the estimated splines. Figure 3.43 illustrates main effects in fitted model 3.6, namely $\hat{f}_1$ (panel (a)), $\hat{f}_2$ (panel (e)), $\hat{f}_{11}$ (panel (b)), $\hat{f}_4$ (panel (c)) and $\hat{f}_6$ (panel (d)). They all turn out simple; of course, the tensor product splines modelling their spatial variation are more complicated. Transects plot like Figure 3.34 can be produced, but they are of little interest. It is more interesting to visualize $\hat{f}_8$, the interaction between two temperature variables. Figure 3.44 visualizes this 3D surface via a perspective plot. The surface is much more complicated compared with the Figure 3.11 from *Manchester 11* case study. This demonstrates the impact of sample size on estimation.

In spatial-temporal modelling residuals should also be check to see if there is unmodelled spatial autocorrelation. Similar to Figure 3.31, I proceed to produce the empirical variograms for spatial residuals in each Monday. Figure 3.45 presents those 52 variograms. On most of the days the fitted model is adequate, leaving no spatial autocorrelation in residuals. But it does leave an excess amount of autocorrelation on day 121.

3.5.4 Removing spatial autocorrelation in residuals with three-way interaction

It is not impossible to add more components to model 3.6 to remove spatial autocorrelation in its residuals. Since the residuals have unmodelled spatial structure, the new components are likely to involve $\{\mathtt{e_i},\ \mathtt{n_i}\}$. However, given that the marginal effect $f_2(\{\mathtt{e_i},\ \mathtt{n_i}\})$ and all two-way interactions $f_3(\mathtt{w_y},\ \{\mathtt{e_i},\ \mathtt{n_i}\})$, $f_5(\mathtt{T^0_{id}},\ \{\mathtt{e_i},\ \mathtt{n_i}\})$ and $f_7(\mathtt{T^*_{id}},\ \{\mathtt{e_i},\ \mathtt{n_i}\})$ between $\{\mathtt{e_i},\ \mathtt{n_i}\}$ and other variables are already in model 3.6, the new components are potentially three-way interactions.

My initial guess was to add a three-margin tensor product spline between $\mathtt{T^0_{id}}$, $\mathtt{T^*_{id}}$ and $\{\mathtt{e_i},\ \mathtt{n_i}\}$, but the resulting model does not remove spatial autocorrelation in residuals. I then turned to the following model

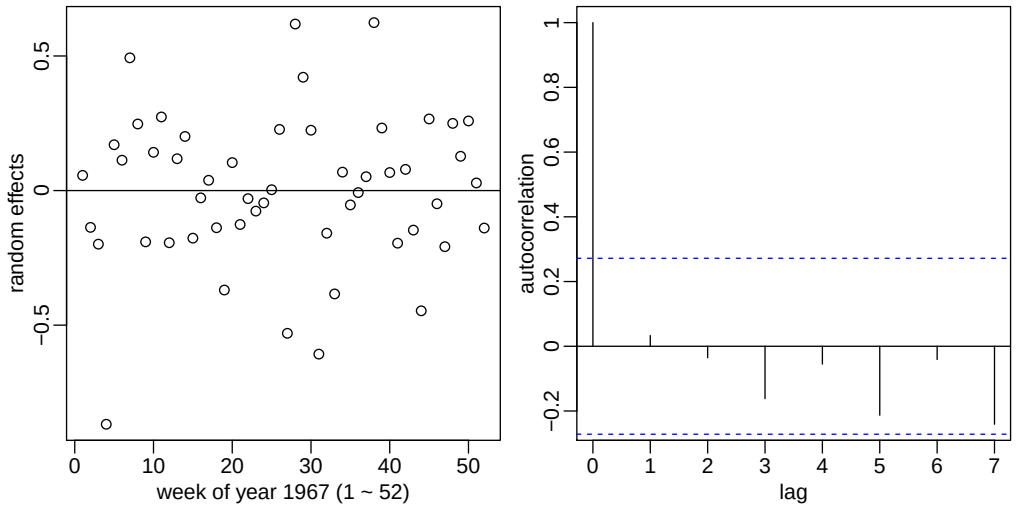


Figure 3.41: Validating i.i.d. assumption for weekly random effects in model 3.6. The left panel is a histogram of predicted random effects and the right panel is their sample autocorrelation. No temporal autocorrelation is spotted.

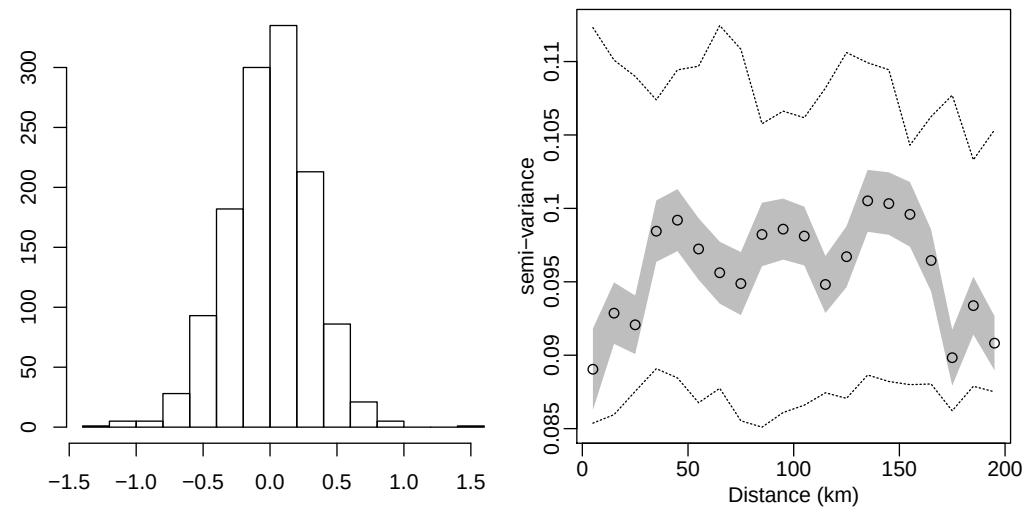


Figure 3.42: Validating i.i.d. assumption for station-specific random effects in model 3.6. The left panel is a histogram of predicted random effects and the right panel is their empirical variogram. No spatial autocorrelation is spotted.

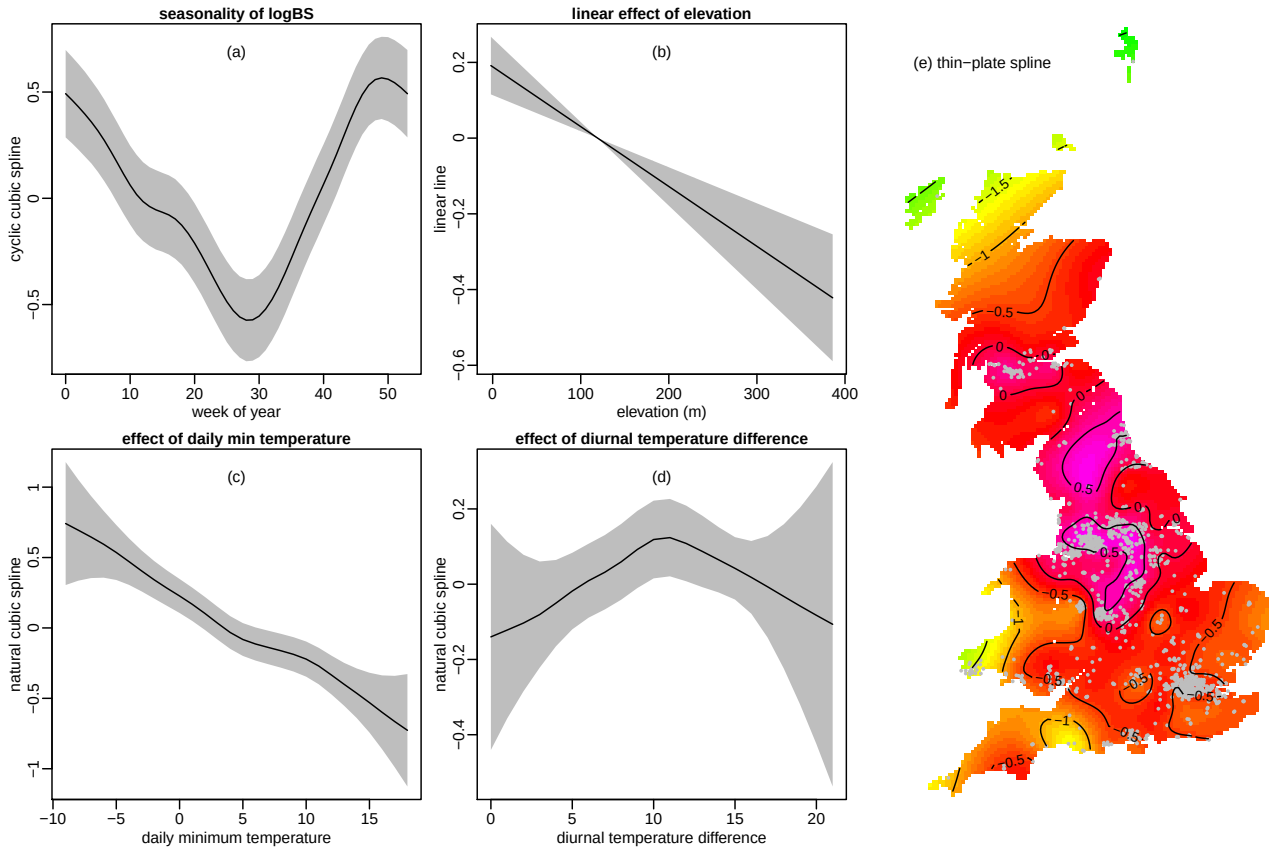


Figure 3.43: Estimated main effects in fitted model 3.6, namely $\hat{f}_1$ (panel (a)), $\hat{f}_2$ (panel (e)), $\hat{f}_{11}$ (panel (b)), $\hat{f}_4$ (panel (c)) and $\hat{f}_6$ (panel (d)).

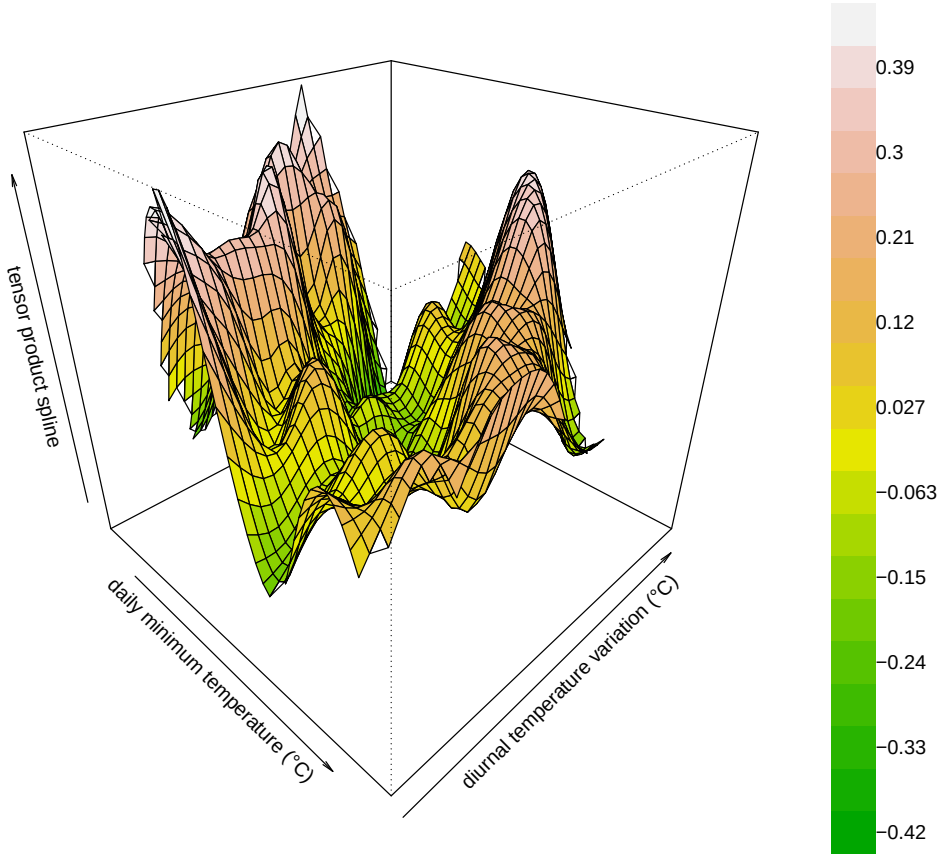


Figure 3.44: A perspective plot to visualize the tensor product spline $\hat{f}_8$ in fitted model

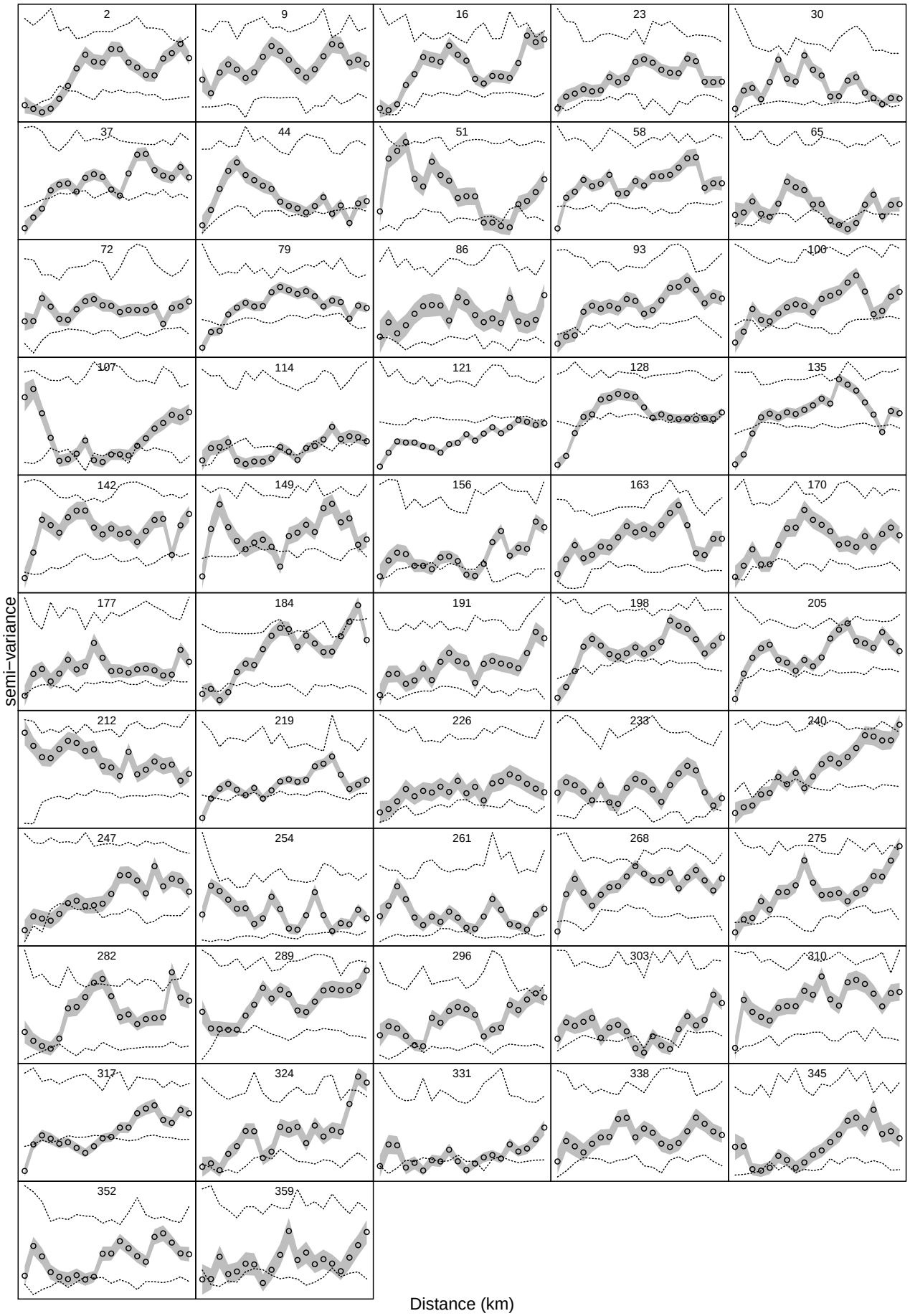


Figure 3.45: Empirical variograms of spatial residuals from fitted model 3.6 on each of the 52 Mondays in year 1967. Axes are omitted for compact display. The variogram is computed up to a distance of 200km at every 5km bin starting from 0 (which is consistent with Figure 3.31). The day of year ($1 \sim 365$) is labelled on each plot.

Table 3.10: Summary of fitted model 3.7 to each day of week in 1967.

(a) n is the number of data; p is the number of regression coefficients; *edf* is the effective degree of freedom of the fitted model; $\hat{\sigma}^2_{\texttt{iw}}$ is the estimated residual variance; $\hat{\sigma}^2_9$ is the estimated variance for station-specific i.i.d. random effect $\hat{f}_9$; $\hat{\sigma}^2_{10}$ is the estimated variance for week of year i.i.d. random effect $\hat{f}_{10}$; adj. R^2 is the adjusted R-squared of the fitted model.

	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
n	54386	54581	55874	56612	56006	54437	55357
p	10348	10348	10347	10347	10347	10346	10349
edf	3918.89	4090.49	4046.72	4132.20	4270.01	4045.32	3893.77
$\hat{\sigma}^2_{\texttt{iw}}$	0.1837	0.1962	0.1923	0.1999	0.1910	0.1968	0.1869
adj. R^2	85.47%	83.89%	84.05%	83.02%	82.40%	84.16%	85.43%
$\hat{\sigma}^2_9$	0.1142	0.1211	0.1142	0.1154	0.1102	0.1069	0.1203
$\hat{\sigma}^2_{10}$	0.0957	0.0747	0.1303	0.1403	0.1489	0.1408	0.1145

(b) Termwise number of regression coefficients p and effective degree of freedom.

	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
$\hat{f}_0$ ($p = 5$)	3.98	3.97	3.97	3.97	3.97	3.97	3.98
$\hat{f}_1$ ($p = 8$)	4.28	4.35	4.03	2.87	2.56	3.31	3.28
$\hat{f}_2$ ($p = 149$)	95.46	94.91	98.26	93.93	103.10	100.06	93.56
$\hat{f}_3$ ($p = 2682$)	1294.29	1328.91	1344.74	1287.46	1461.13	1355.35	1206.16
$\hat{f}_4$ ($p = 9$)	4.17	5.69	2.73	5.38	5.80	5.24	3.28
$\hat{f}_5$ ($p = 891$)	196.16	202.92	255.95	230.91	286.09	246.49	212.08
$\hat{f}_6$ ($p = 9$)	1.44	1.24	1.00	1.00	1.00	4.00	4.23
$\hat{f}_7$ ($p = 891$)	210.96	243.02	173.71	260.18	202.78	221.31	232.16
$\hat{f}_8$ ($p = 196$)	56.01	61.43	57.07	60.98	39.05	35.45	49.98
$\hat{f}_9$ ($p = 1275$)	1119.04	1119.58	1115.44	1117.59	1109.85	1106.46	1123.40
$\hat{f}_{10}$ ($p = 52$)	46.25	46.05	46.63	47.75	48.07	47.33	48.35
$\hat{f}_{11}$ ($p = 4$)	1.00	1.00	1.00	1.00	1.00	1.00	1.00
$\hat{f}_{12}$ ($p = 2088$)	421.92	503.85	468.50	531.82	490.31	441.06	442.65
$\hat{f}_{13}$ ($p = 2088$)	462.90	472.57	472.69	486.35	514.29	473.28	468.66

Model 3.7: model 3.6 + three-way interaction

$$\begin{aligned}\texttt{logBS}_{\texttt{iw}} = {} & f_0(\texttt{E}_\texttt{i}) + f_1(\texttt{w}_\texttt{y};\ 10) + f_2(\{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\};\ 150) + f_3(\texttt{w}_\texttt{y},\ \{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\};\ 20,\ 150) + \\ & f_4(\texttt{T}^0_\texttt{id};\ 10) + f_5(\texttt{T}^0_\texttt{id},\ \{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\};\ 10,\ 100) + \\ & f_6(\texttt{T}^*_\texttt{id};\ 10) + f_7(\texttt{T}^*_\texttt{id},\ \{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\};\ 10,\ 100) + \\ & f_8(\texttt{T}^0_\texttt{id},\ \texttt{T}^*_\texttt{id};\ 15,\ 15) + f_9(\texttt{i};\ 1275) + f_{10}(\texttt{w}_\texttt{y};\ 52) + f_{11}(\texttt{h}_\texttt{i};\ 5) + \\ & f_{12}(\texttt{w}_\texttt{y},\ \texttt{T}^0_\texttt{id},\ \{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\};\ 10,\ 10,\ 30) + f_{13}(\texttt{w}_\texttt{y},\ \texttt{T}^*_\texttt{id},\ \{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\};\ 10,\ 10,\ 30) + \epsilon_{\texttt{iw}},\end{aligned}$$

where two three-way interactions f_{12} and f_{13} between space, time and temperature are added. Note that compared with model 3.6, the k values of many terms have been reduced. In particular, the k value for $\{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\}$ margin in f_{12} and f_{13} is set very low. Whether this is reasonable or not, it is the only way to be able to practically fit this model. Model 3.6 readily has more than 10000 coefficients, and if k values are not reduced, this new model would involve 110000+ coefficients, even more than the number of data (54386). After reducing k, this model still has about 10000 parameters. This model turns out to successfully remove all spatial autocorrelation in residuals (see Figure 3.46). I have also fitted this model to logBS from other days of week, and without exception, no spatial autocorrelation is seen from spatial residuals (see Figures 3.47, 3.48, 3.49, 3.50, 3.51, 3.52). Table 3.10 is a summary of those seven fitted models.

However, it is hard to interpret these three-way interaction. Basically they mean that temperature effects can change at any location and any time (hence essentially saying that there is no relationship between temperature and logBS?). In fact, if you think about this carefully, it is not difficult to understand why these components are so powerful to remove spatial autocorrelation. Temperature

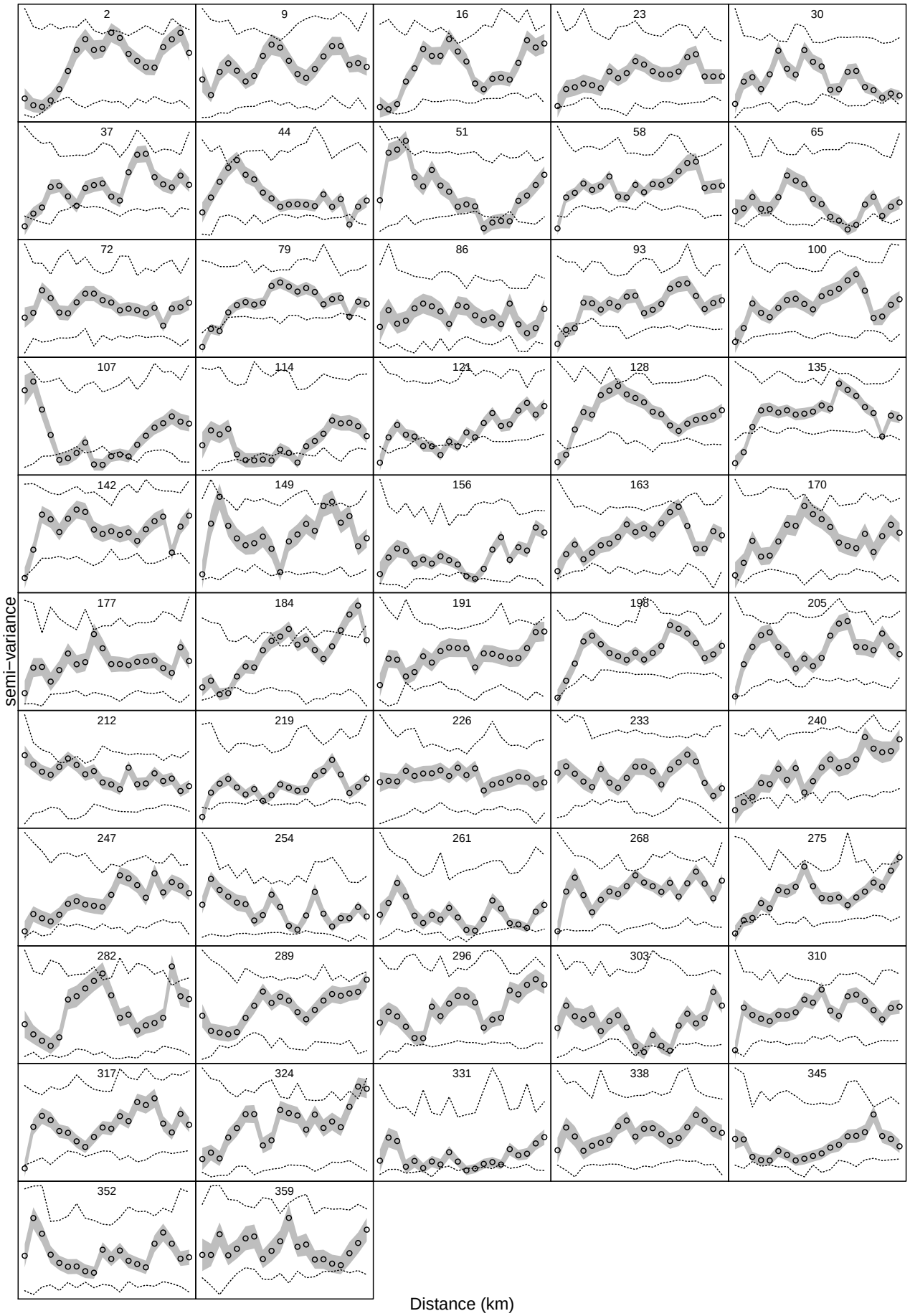


Figure 3.46: Empirical variograms of spatial residuals from fitted model 3.7 on each of the 52 **Mondays** in year 1967. Compared with Figure 3.46, the three-way interactions have helped remove all spatial autocorrelation in residuals.

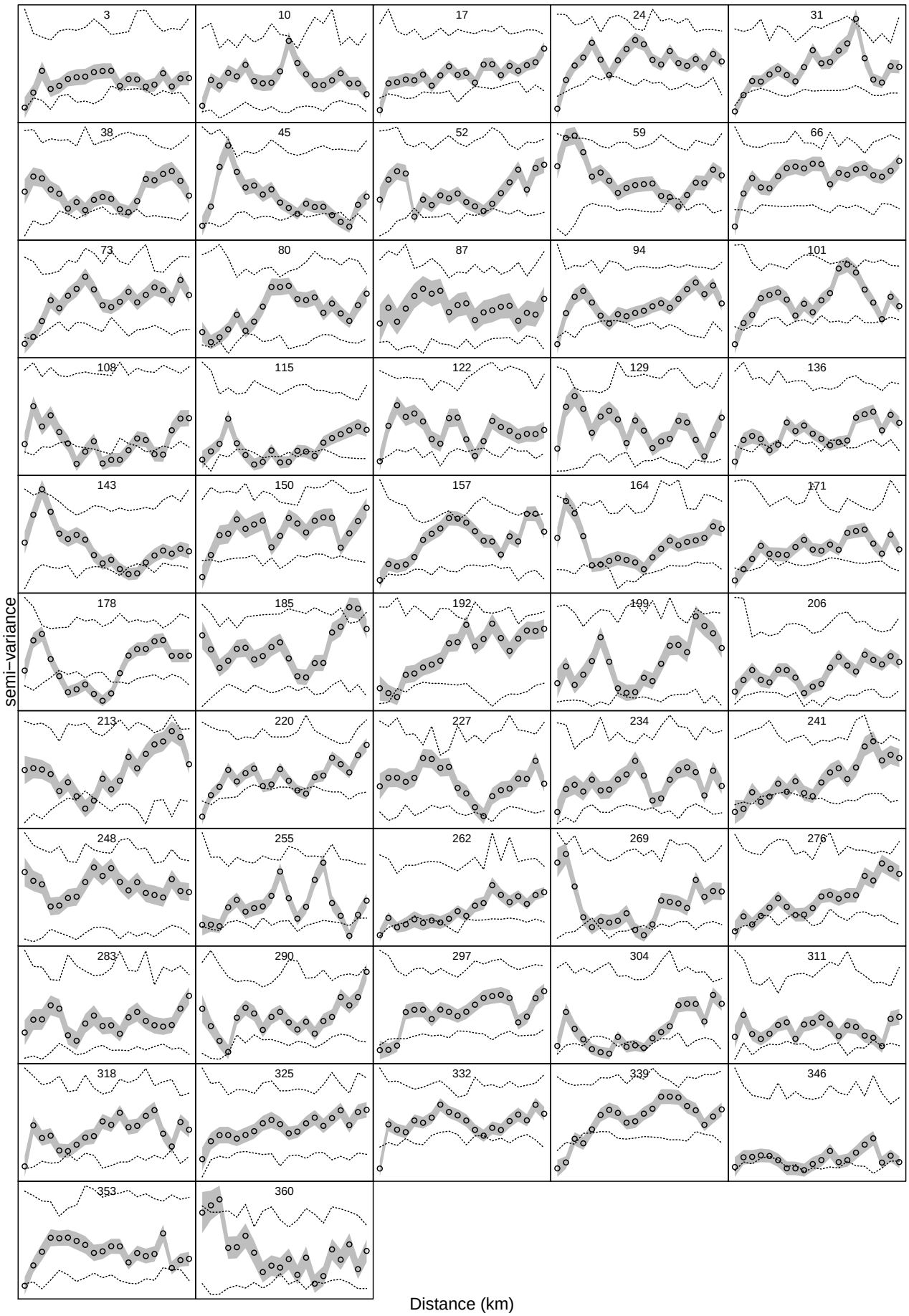


Figure 3.47: Empirical variograms of spatial residuals from fitted model 3.7 on each of the 52 **Tuesdays** in year 1967.

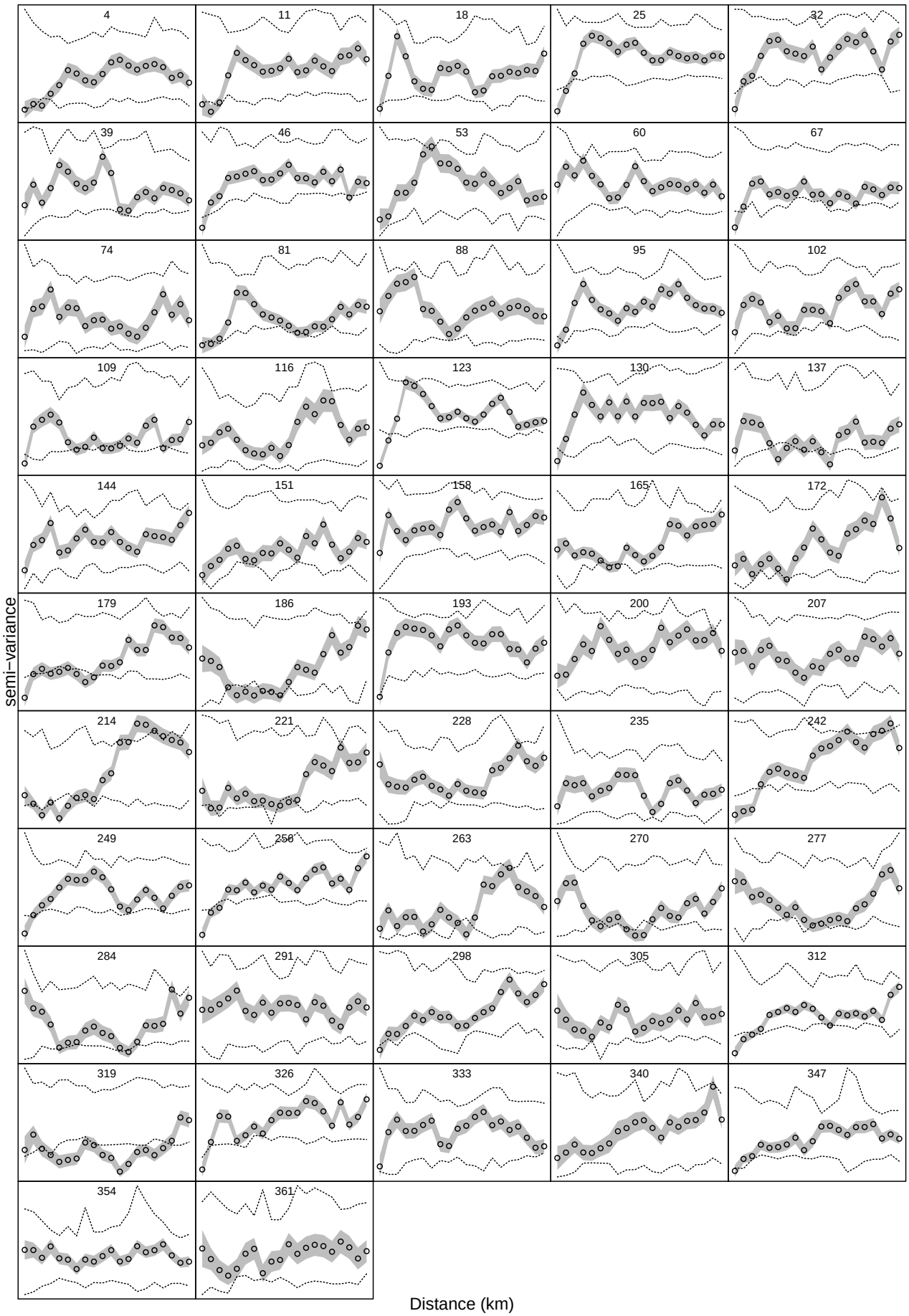


Figure 3.48: Empirical variograms of spatial residuals from fitted model 3.7 on each of the 52 **Wednesdays** in year 1967.

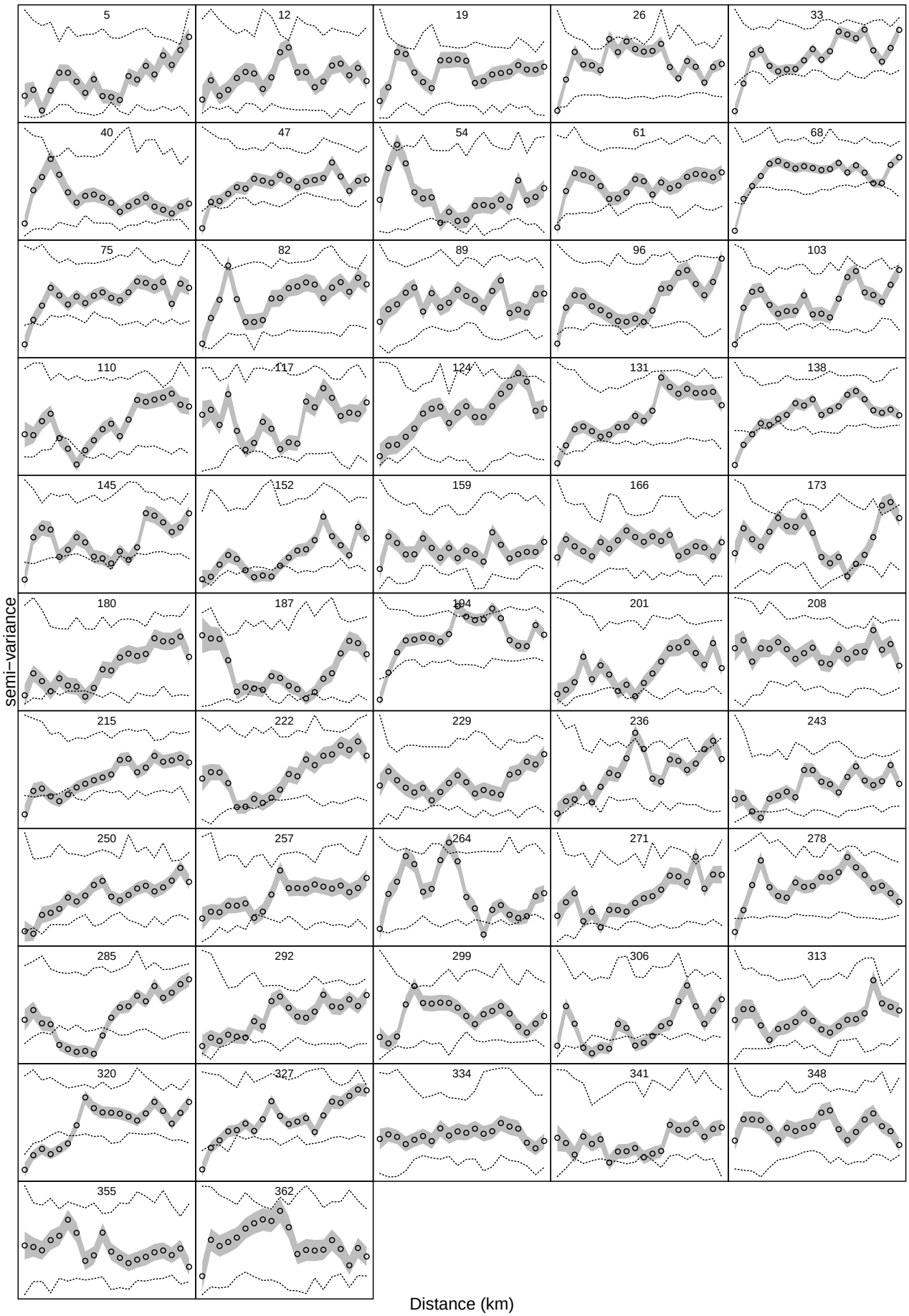


Figure 3.49: Empirical variograms of spatial residuals from fitted model 3.7 on each of the 52 **Thursday** in year 1967.

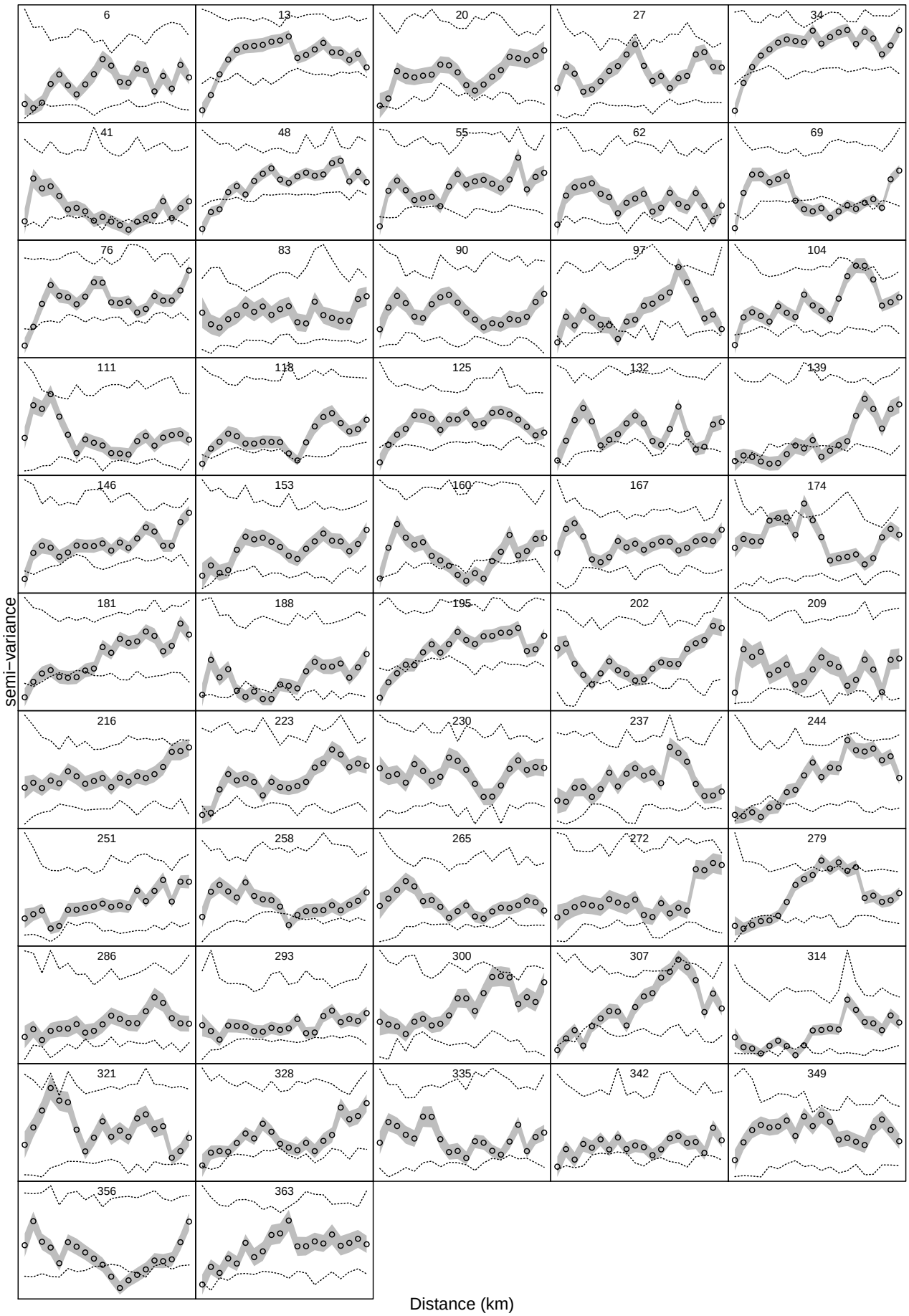


Figure 3.50: Empirical variograms of spatial residuals from fitted model 3.7 on each of the 52 **Friday** in year 1967.

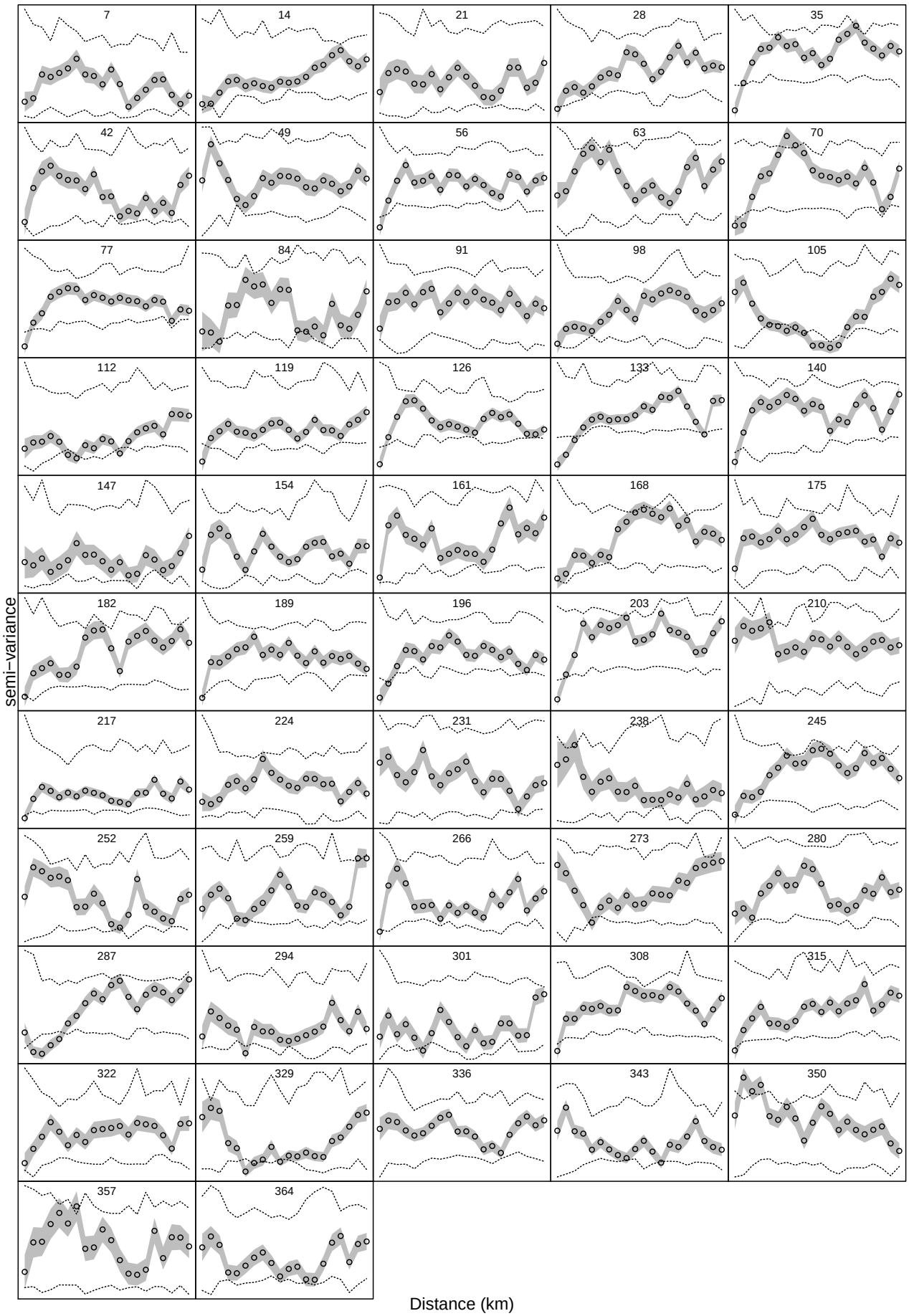


Figure 3.51: Empirical variograms of spatial residuals from fitted model 3.7 on each of the 52 **Saturdays** in year 1967.

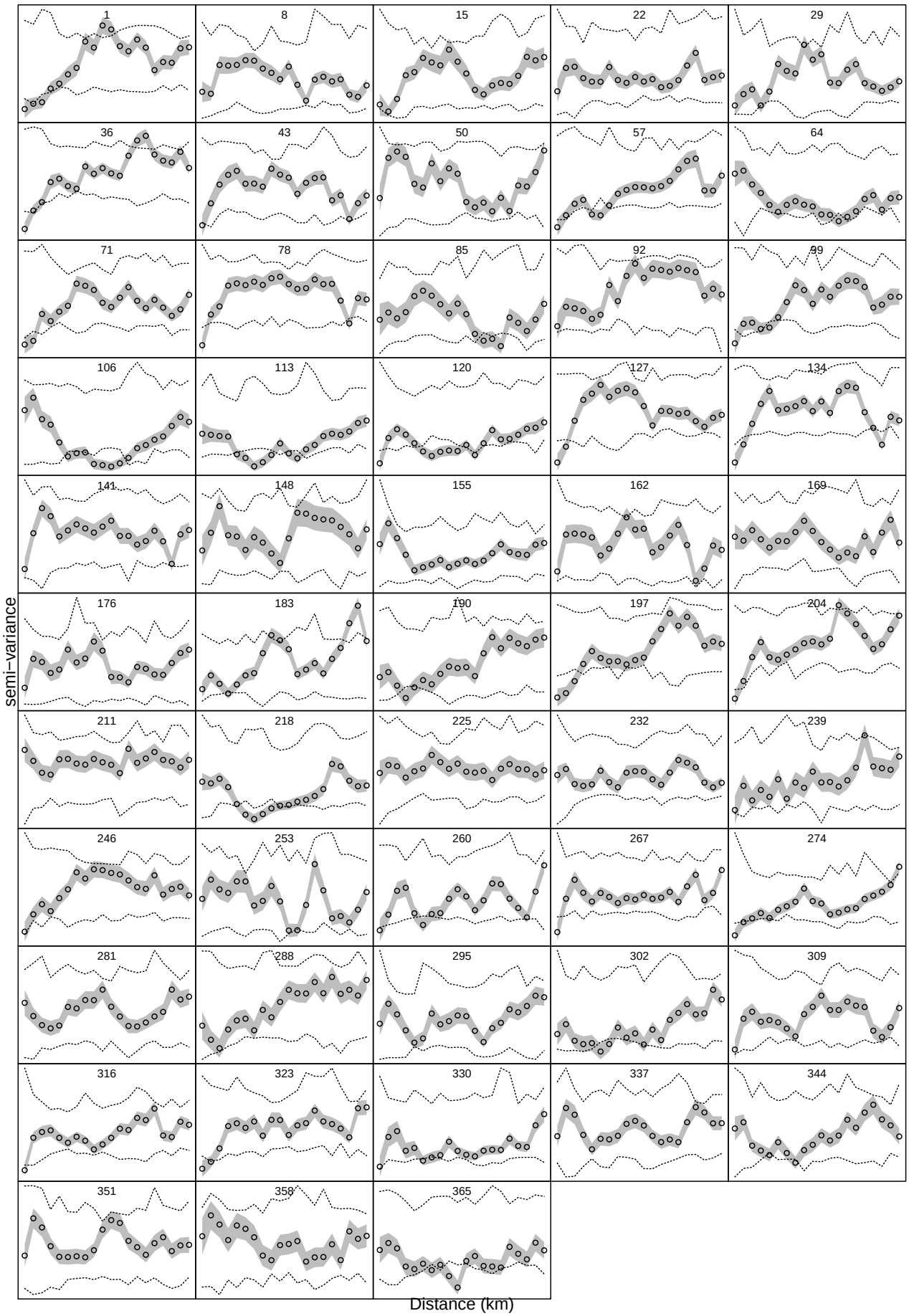


Figure 3.52: Empirical variograms of spatial residuals from fitted model 3.7 on each of the 53 **Sundays** in year 1967.

Table 3.11: Number of data n, model complexity p (number of parameters) and model fitting time T (in seconds) for logBS models so far.

Manchester 11 model 3.3b			annual mean logBS model 3.5			Monday logBS in 1967 model 3.6		
n	p	T	n	p	T	n	p	T
15707	732	442.15	24239	6750	9927.68	54386	10932	45962.61

varables are the only daily variables that are available in my Black Smoke dataset. Even a fake linear model like

$$\texttt{logBS}_{\texttt{id}} = \alpha^0_{\texttt{id}}\texttt{T}^0_{\texttt{id}} + \alpha^*_{\texttt{id}}\texttt{T}^*_{\texttt{id}} + \epsilon_{\texttt{id}}$$

would be able to well "predict" logBS if the coefficients $\alpha^0_{\texttt{id}}$ and $\alpha^*_{\texttt{id}}$ can vary freely over space and time. So I am very skeptical of model 3.7.

The need for these odd-looking three-way interactions is probably a hint that other daily covariates than temperatures are necessary for modelling daily logBS. However, this can not be possibly done in this PhD research (acquiring daily temperature from UKCP09 had already been a non-trivial task).

Should these three-way interactions be used for model development? A possible way to answer this is running cross-validation. However, this is far beyond the computational capability that current GAM fitting method could offer.

3.5.5 Encountering computational hurdle

Previous presentation on model development was made so smoothly, as if they had been easy to obtain. The true story is, model fitting was very slow. Practical model building was further slowed down by the non-negligible costs for choosing k and updating a present model with new components. Table 3.11 tells you how much time was spent for fitting model 3.3b for *Manchester 11*, model 3.5 for annual mean logBS and model 3.6 for Monday logBS in 1967. The number of data and number of parameters / coefficients are also provided for a reference. Fitting time for model 3.7 is not included. It has similar n and p to model 3.6 hence similar fitting time. Note that fitting these daily models takes half a day each. It had taken nearly a week to produce the results in Table 3.10 and Figures 3.46 to 3.52. So it is time to do something to speed up model fitting, otherwise no further models can be feasibly built for Black Smoke.

3.6 Summary

The model development for logBS in this Chapter, on one hand, offers solid case studies for time series modelling and spatial-temporal modelling via GAM, and on the other hand, reveals the high complexity required for building GAMs for daily logBS even just from a single year. Many statistical questions can be raised regarding the model development, like

1. What is the reason behind the spatial-temporal extrapolation problem in model 3.5 and is there any way to suppress it?
2. Should three-way interactions in model 3.7 be used to adequately model space-time relationship? Won't they actually lead to overfitting?

But they all seem to give way to the computational hurdle in fitting "big" models: models with about 10^5 data and parameters.

It is time to dive into the computational engine of additive models and improve its design so that model estimation with big datasets and high complexity is feasible and more importantly, as efficient as possible.

To see opportunities in upgrading GAM computational engine requires in the first place a solid understanding of the exsiting one. So the next Chapter will review all computational details for GAM with large datasets. It is also at this point that the flavour of this thesis is going to change. There will be fewer nice looking pictures; instead, there will be more complicated mathematical notations, algorithms and benchmarking.

Chapter 4

GAM computation for large datasets: a review

This Chapter formally introduces the computational details underneath the `bam` function of `R` package `mgcv` (prior to `mgcv_1.8-7`), which is designed for estimating a GAM with large datasets.

GAM computations are ultimately about matrix computations, so from now on, issues of matrix computations are recurring. To ease later demonstrations, §4.1 will offer some preliminaries on the most frequently referenced matrix computations in this thesis.

Methods for large datasets was first developed for additive models. It breaks down the model fitting process into two stages: model matrix reduction (to be introduced in §4.2) and REML estimation (to be introduced in §4.3). Methods for generalized additive models is an iterative application of such method for additive models. Such way of estimating a GAM is termed "performance iteration". There is another way for GAM estimation called "outer iteration". It uses a different logic on iterations and is implemented in the `gam` function of `mgcv`. I will in §4.4 give a brief introduction to this method, pointing out its key difference from "performance iteration". Computational details will not be provided as the method is infeasible for large datasets.

These computational details in this Chapter are largely based on Wood (2011) and Wood et al. (2015), but integrated together for a coherent presentation on estimation of additive models. Materials in this Chapter should be a very good reference for anyone who wants to dive into `bam`. The two papers essentially use different estimation methods: the former is implemented in `gam` function using "outer iteration" and is not "bigdata" oriented. The latter one mainly demonstrates the model matrix reduction idea, and simply cites the former for derivative computation in REML estimation. However, while they do share something similar, the derivative computation is still substantially different.

I would also like to give a pre-notice, that my demonstration for REML estimation in §5 is not 100% "loyal" to current `bam` implementation. In the learning process of the computational engine, I discovered a few places where the design idea could be better implemented. These actually contribute to my first round of performance optimization for `bam`, to be benchmarked in the next Chapter. In this Chapter, I will just present my improved version of REML estimation; the inefficiency in the original version will be pointed out in the next Chapter.

4.1 Preliminaries on matrix computations

This section gives a brief introduction to some important matrix computations in the heart of scientific computing.

- §4.1.1 introduces Householder transformation, Householder QR factorization, the mathematical meaning of QR factorization, the "factored form storage" of factor 'Q', and matrix multiplication involving factor 'Q'.
- §4.1.2 introduces Cholesky factorization and its link with QR factorization.
- §4.1.3 and §4.1.4 introduce some computations associated with triangular matrices, namely solving a triangular system of linear equations, inverting a triangular matrix and computing cross-product of a triangular matrix. These operations are essential since both QR factorization and Cholesky factorization produce a triangular matrix factor.
- §4.1.5 introduces a subtype of general eigen decomposition: the symmetric eigen decomposition for a real symmetric matrix.

I will not demonstrate algorithms for all computations. The aim is to provide background information for their practical applications in GAM computations to assist you in understanding subsequent sections in this Chapter. Basic knowledge of linear algebra like matrix multiplication, matrix inverse, positive-definite matrix and orthogonal matrix (rotation matrix or reflection matrix) is a prerequisite.

Numerical matrix computations primarily consist of floating-point arithmetic, specially floating-point multiplication and addition, hence their computational complexity is often measured by the amount of multiplication and addition they involve, known as the number of floating-point operations (FLOP). For example, matrix multiplication between an $m \times k$ matrix $\boldsymbol{A}$ and a $k \times n$ matrix $\boldsymbol{B}$ involves $2mkn$ FLOP, since 1) the resulting matrix has mn elements; 2) element $(i,\ j)$ is computed by a dot product $\sum_{l=1}^{k} \boldsymbol{A}(i,\ l)\boldsymbol{B}(l,\ j)$ which invovles k addition and k multiplication. FLOP count for all matrix computations covered in this section will be given along the way, but is also summarized in §4.1.6. The summary section also gives numerous fruitful examples on how matrix factorizations are useful in practice. You may have a quick glimpse on that section first to gain extra motivation to go through the following possibly tedious pages.

4.1.1 Householder QR factorization

There are several types of QR factorization, but a particular type known as Householder QR factorization is applied throughout this thesis.

A Householder matrix is a reflection matrix $\boldsymbol{H} = \boldsymbol{I} - \tau \boldsymbol{v}\boldsymbol{v}'$, where $\boldsymbol{v}$ is a non-zero vector called a reflector vector and $\tau = \frac{2}{\|\boldsymbol{v}\|^2}$. Given two length-$n$ vectors $\boldsymbol{x}$ and $\boldsymbol{y}$, there exists a Householder transformation $\boldsymbol{H}\boldsymbol{x} = \boldsymbol{y}$, where $\boldsymbol{v} = \boldsymbol{x} - \boldsymbol{y}$. A special type of Householder transformation is

$$\boldsymbol{H} : \begin{bmatrix} \boldsymbol{x}(1) \\ \boldsymbol{x}(2:n) \end{bmatrix} \to \begin{bmatrix} \boldsymbol{y}(1) \\ \boldsymbol{0} \end{bmatrix}, \quad \boldsymbol{y}(1) = -\mathtt{sgn}(\boldsymbol{x}(1))\sqrt{\boldsymbol{x}(1)^2 + \|\boldsymbol{x}(2:n)\|^2}.$$

Such type of transformation can be sequentially applied to zero the lower triangular part of a matrix, yielding an upper triangular matrix. See Figure 4.1 for an illustration.

In general, let $\boldsymbol{X}$ be an $n \times p$ matrix ($n \geq p$), p sequential Householder transformations to $\boldsymbol{X}$ yields an upper triangular matrix

$$\boldsymbol{H_p}\boldsymbol{H_{p-1}} \cdots \boldsymbol{H_1}\boldsymbol{X} = \boldsymbol{\mathcal{R}}.$$

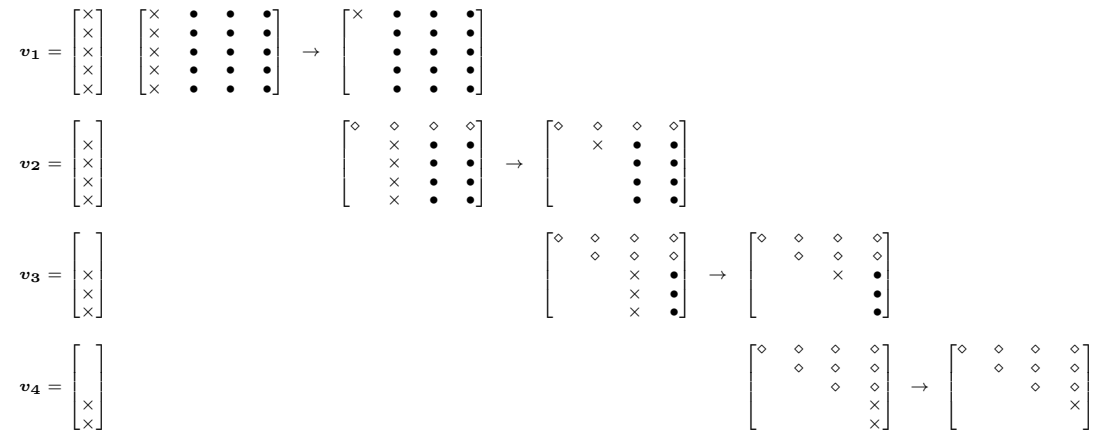


Figure 4.1: Illustration of Householder QR factorization using a 5×4 matrix. Four Householder transformations are needed to arrive at a final upper triangular matrix. Zero entries in matrices and vectors are left blank. Place holders $\bullet$, $\times$ and $\diamond$ represent non-zero entries, with the following meanings: $\times$ are used to derive reflector vectors; $\bullet$ are updated by the resulting Householder transformation; $\diamond$ are unaffected as the "range" of the transformation is shrinking toward the lower right submatrix at each step.

These transformations can be summarized by a single orthonormal transformation $\boldsymbol{\mathcal{Q}}'\boldsymbol{X} = \boldsymbol{\mathcal{R}}$, or $\boldsymbol{X} = \boldsymbol{\mathcal{Q}}\boldsymbol{\mathcal{R}}$, which is a QR factorization. Note that

- The $n \times n$ matrix $\boldsymbol{\mathcal{Q}} = \boldsymbol{H_1 H_2 \cdots H_p}$ is mostly useful for paper work. It needs not be formed and is practically stored in a "factored form" as a number of reflector vectors. Later I will explain the storage of such "factored form" and how matrix multiplication involving $\boldsymbol{\mathcal{Q}}$ is practically done;
- The $\boldsymbol{\mathcal{R}}$ matrix above is not a triangular matrix, but its upper $p \times p$ submatrix is. In many theoretical derivations, QR factorization is often written in a partitioned or "thin" form

$$\boldsymbol{X} = \begin{pmatrix} \boldsymbol{Q} & \boldsymbol{Q}^\perp \end{pmatrix} \begin{pmatrix} \boldsymbol{R} \\ \boldsymbol{0} \end{pmatrix} = \boldsymbol{QR},$$

  where $\boldsymbol{Q} = \boldsymbol{\mathcal{Q}}(,\ 1:p)$, $\boldsymbol{Q}^\perp = \boldsymbol{\mathcal{Q}}(,\ (p+1):n)$ and $\boldsymbol{R} = \boldsymbol{\mathcal{R}}(1:p,\ )$.

For more details on Householder transformation and Householder QR factorization, see (Golub and Loan, 2013, §5.1 and §5.2).

If $\boldsymbol{A}$ has full column rank, then finding an orthonormal basis for the column space of $\boldsymbol{A}$ is a Gram-Schmidt process. In fact, QR factorization is the matrix representation for Gram-Schmidt process (just like LU factorization is that for Gaussian elimination), and the columns of $\boldsymbol{Q}$ is just the desired basis. At the same time, columns of $\boldsymbol{Q}^\perp$ is an orthonormal basis for the null space (or kernel) of $\boldsymbol{A}'$ (A more concrete explanation will be made in §4.1.6).

But such nice interpretation of QR factorization does not hold when $\boldsymbol{A}$ does not have full column rank. A column pivoting strategy is used to overcome this and support rank estimation. The resulting thin-QR factorization has the form

$$\boldsymbol{AP} = \boldsymbol{QR},$$

where $\boldsymbol{P}$ is a column permutation matrix (a variant of an identity matrix by column permutation), and $\boldsymbol{R}$ is an upper triangular matrix with $|\boldsymbol{R}(1,\ 1)| \geq |\boldsymbol{R}(2,\ 2)| \geq \cdots \geq |\boldsymbol{R}(p,\ p)| \geq 0$. In practice, $\boldsymbol{P}$ is simply stored as an index vector $\boldsymbol{k}$, so that $(\boldsymbol{AP})(,\ i) = \boldsymbol{A}(,\ \boldsymbol{k}(i))$. Golub and Loan (2013, §5.4.2) gives algorithm details for such factorization. Note that the factorization helps supporting rank estimation but is not directly rank-revealing. While it is tempting to use $|\boldsymbol{R}(i,\ i)|\ /\ |\boldsymbol{R}(1,\ 1)| < \epsilon$ for rank estimation, where ϵ is some numerical tolerance, Golub and Loan (2013, §5.4.3) gives a counterexample. The condition estimation described in Golub and Loan (2013, §3.5.4) can serve for

rank estimation. If $\boldsymbol{R}$ has an estimated rank of r, then columns of $\boldsymbol{Q}(,\ 1:r)$ is an orthonormal basis of the column space of $\boldsymbol{A}$.

QR factorization with column pivoting (hereafter called "pivoted QR factorization" for short) can be seen as a QR factorization with a non-triangular 'R' factor. A column permutation matrix is invertible, and in fact is orthogonal with $\boldsymbol{P}^{-1} = \boldsymbol{P}'$. Therefore,

$$\boldsymbol{AP} = \boldsymbol{QR} \ \Rightarrow \ \boldsymbol{A} = \boldsymbol{QR}^*,\ \boldsymbol{R}^* = \boldsymbol{RP}'.$$

Applying $\boldsymbol{P}'$ to a matrix from the right is reversing the column permutation: $\boldsymbol{R}^*(,\ \boldsymbol{k}(i)) = \boldsymbol{R}(,\ i)$. Later in this thesis, a thin QR factorization may be expressed by $\boldsymbol{A} = \boldsymbol{QR}^*$ for briefness, and you should remember that $\boldsymbol{R}^*$ in this case absorbs the reverse column permutation.

QR factorization has a computational complexity of $(2np^2 - \frac{2}{3}p^3)$ FLOP. Some literature dealing with "tall-thin" matrices with $n \gg p$, often put it $2np^2$. However, if $n = p$, the complexity is $\frac{4}{3}p^3$. Pivoting would add some $O(np)$ overhead, but it is ignorable.

You may have already spotted from Figure 4.1 that (non-zero entries of) the reflector vectors $\boldsymbol{v_i}$ can be stored into the lower trapezoidal part of $\boldsymbol{A}$, so that in the end of the factorization, the upper triangular part of $\boldsymbol{A}$ gives $\boldsymbol{R}$, while the lower trapezoidal part holds the sequence of $\boldsymbol{v_i}$. However, both $\boldsymbol{v_i}$ and $\boldsymbol{R}$ demand the diagonal of $\boldsymbol{A}$. Such clash can be avoided if $\boldsymbol{v_i}$ is rescaled so that $\boldsymbol{v_i}(i) = 1$, then there is no need to store those 1. Such storage is called the "factored form storage" of $\boldsymbol{\mathcal{Q}}$.

Matrix multiplications involving $\boldsymbol{\mathcal{Q}}$, namely $\boldsymbol{\mathcal{Q}'B}$, $\boldsymbol{\mathcal{Q}B}$, $\boldsymbol{B\mathcal{Q}'}$ and $\boldsymbol{B\mathcal{Q}}$, should be done by sequential Householder transformations. As $\boldsymbol{B\mathcal{Q}'} = \boldsymbol{(\mathcal{Q}B')'}$ and $\boldsymbol{B\mathcal{Q}} = \boldsymbol{(\mathcal{Q}'B')'}$, it is sufficient to demonstrate the computation of left-side orthonormal transformations $\boldsymbol{\mathcal{Q}'B}$ and $\boldsymbol{\mathcal{Q}B}$. For example, orthornormal transformation $\boldsymbol{\mathcal{Q}'B}$ is just $\boldsymbol{H_pH_{p-1}\cdots H_1B}$. Starting at $\boldsymbol{B_0} = \boldsymbol{B}$, we iteratively compute $\boldsymbol{B_i} = \boldsymbol{H_iB_{i-1}}$, and in the end $\boldsymbol{\mathcal{Q}'B} = \boldsymbol{B_p}$. Aslo note that there is even no need to form $\boldsymbol{H_i}$, as $\boldsymbol{H_iB} = (\boldsymbol{I} - \tau_i\boldsymbol{v_iv_i'})\boldsymbol{B} = \boldsymbol{B} - \tau_i\boldsymbol{v_i}(\boldsymbol{B'v_i})'$ so the computation is not about matrix-matrix multiplication at all. Finally, the structure that $\boldsymbol{v_i}(1:(i-1)) = \boldsymbol{0}$ can be exploited, so that $\boldsymbol{H_iB}$ would leave $\boldsymbol{B}(1:(i-1),\ )$ unchanged. This explains why the "range" of Householder transformation shrinks in Figure 4.1 as the factorization proceeds.

Matrix multiplication involving $\boldsymbol{Q}$ or $\boldsymbol{Q}^\perp$ needs a post-processing or pre-processing, as

$$\begin{aligned}
\boldsymbol{\mathcal{Q}'B} &= \begin{bmatrix}\boldsymbol{Q} & \boldsymbol{Q}^\perp\end{bmatrix}'\boldsymbol{B} = \begin{bmatrix}\boldsymbol{Q'B} \\ \boldsymbol{Q}^{\perp\prime}\boldsymbol{B}\end{bmatrix}, \\
\boldsymbol{QB} &= \begin{bmatrix}\boldsymbol{Q} & \boldsymbol{Q}^\perp\end{bmatrix}\begin{bmatrix}\boldsymbol{B} \\ \boldsymbol{0}\end{bmatrix} = \boldsymbol{\mathcal{Q}}\begin{bmatrix}\boldsymbol{B} \\ \boldsymbol{0}\end{bmatrix}, \\
\boldsymbol{Q}^\perp\boldsymbol{B} &= \begin{bmatrix}\boldsymbol{Q} & \boldsymbol{Q}^\perp\end{bmatrix}\begin{bmatrix}\boldsymbol{0} \\ \boldsymbol{B}\end{bmatrix} = \boldsymbol{\mathcal{Q}}\begin{bmatrix}\boldsymbol{0} \\ \boldsymbol{B}\end{bmatrix}.
\end{aligned}$$

Thus for example, $\boldsymbol{Q'B} = (\boldsymbol{\mathcal{Q}'B})(1:p,\ )$ and $\boldsymbol{QB}$ is an orthonormal transformation on a zero padded $\boldsymbol{B}$.

4.1.2 Cholesky factorization

Cholesky factorization is a special case of LU factorization, which is a triangular matrix factorization for a full-rank square matrix. Let $\boldsymbol{S}$ be a $p \times p$ positive-definite matrix, its Cholesky factorization is

$$\boldsymbol{S} = \boldsymbol{LL'} \quad \text{or} \quad \boldsymbol{S} = \boldsymbol{R'R},$$

where $\boldsymbol{L}$ and $\boldsymbol{R}$ are a lower and upper triangular matrix respectively, often called the lower and upper Cholesky factor of $\boldsymbol{S}$. An understanding of computation algorithm is useful but not essential to follow

this thesis. I refer keen readers to Golub and Loan (2013, §4.2.5) for the algorithm which involves $\frac{1}{3}p^3$ FLOP.

The representation with upper triangular factor is most illuminating for establishing a link between Cholesky factorization and QR factorization, because if $\boldsymbol{A} = \boldsymbol{QR}$, then $\boldsymbol{A'A} = \boldsymbol{R'R}$ so $\boldsymbol{R}$ is "almost" the upper Cholesky factor of $\boldsymbol{A'A}$. I used "almost" as in practice they are only identical up to sign flipping. Cholesky factorization algorithm yields an $\boldsymbol{R}$ matrix with positive diagonal elements, while QR factorization does not, though it can be required(Demmel et al., 2009).

If $\boldsymbol{S}$ is only positive-semidefinite, then the Cholesky factorization can only proceed with pivoting. This analogizes the QR factorization where $\boldsymbol{A}$ does not have full column rank, because then $\boldsymbol{A'A}$ will be positive-semidefinite. The factorization in this case (hereafter called "pivoted Cholesky factorization") has the form

$$\boldsymbol{P'SP} = \boldsymbol{LL'} \quad \text{or} \quad \boldsymbol{P'SP} = \boldsymbol{R'R},$$

where $\boldsymbol{P}$ is a column permutation matrix (applying $\boldsymbol{P'}$ to a matrix from the left is effectively doing row permutation), which is practically stored as an index vector, and $\boldsymbol{R}$ is an upper triangular matrix with $\boldsymbol{R}(1,\ 1) \geq \boldsymbol{R}(2,\ 2) \geq \cdots \geq \boldsymbol{R}(p,\ p) \geq 0$.

$\boldsymbol{S}$ needs better be pre-conditioned for greater numerical stability in the pivoted Cholesky factorization. This is particularly true when we factorize $\boldsymbol{A'A}$ as an analogy to QR factorization of $\boldsymbol{A}$, as $\boldsymbol{A'A}$ has twice as big the conditional number as $\boldsymbol{A}$. A common and simple pre-conditioner is the Jacobian diagonal pre-conditioner. For a symmetric matrix the pre-conditioner is applied from both sides of the matrix. Let $\boldsymbol{J} = \texttt{diag}\big(\sqrt{\texttt{diag}(\boldsymbol{S})}\big)$, the pre-conditioned matrix is $\boldsymbol{J}^{-1}\boldsymbol{S}\boldsymbol{J}^{-1}$. The pivoted Cholesky factorization, for example, is then

$$\boldsymbol{P'}(\boldsymbol{J}^{-1}\boldsymbol{S}\boldsymbol{J}^{-1})\boldsymbol{P} = \boldsymbol{R'R}.$$

Pivoted Cholesky factorization (and optionally with pre-conditioning) can be seen as a Cholesky factorization with non-triangular Cholesky factor, because

$$\begin{aligned} \boldsymbol{P'SP} = \boldsymbol{R'R} \ &\Rightarrow\ \boldsymbol{S} = \boldsymbol{R^{*\prime}R^*},\ \boldsymbol{R^*} = \boldsymbol{RP'}, \\ \boldsymbol{P'}(\boldsymbol{J}^{-1}\boldsymbol{S}\boldsymbol{J}^{-1})\boldsymbol{P} = \boldsymbol{R'R} \ &\Rightarrow\ \boldsymbol{S} = \boldsymbol{R^{*\prime}R^*},\ \boldsymbol{R^*} = \boldsymbol{RP'J}. \end{aligned}$$

Later in this thesis, a Cholesky factorization may be expressed by $\boldsymbol{S} = \boldsymbol{R^{*\prime}R^*}$ for briefness, and you should remember that $\boldsymbol{R^*}$ in this case absorbs the reverse column permutation and pre-conditioning.

4.1.3 Solving a triangular system of linear equations

Taking an upper triangular system $\boldsymbol{Rx} = \boldsymbol{y}$, or

$$\begin{pmatrix} R_{11} & R_{12} & \cdots & R_{1p} \\ & R_{22} & \cdots & R_{2p} \\ & & \ddots & \vdots \\ & & & R_{pp} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_p \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_p \end{pmatrix}$$

as an example, the solution is a back substitution

$$x_i = R_{ii}^{-1}\left(y_i - \sum_{k=i+1}^{p} R_{ik}x_k\right), \quad i = p,\ (p-1),\ \cdots,\ 1.$$

That is, $x_1,\ x_2,\ \cdots, x_p$ can be determined one by one backward. For lower triangular system $\boldsymbol{R'x} = \boldsymbol{y}$ the solution is similarly a forward substitution. Both substitutions cost p^2 FLOP, and they are practically used for computing $\boldsymbol{x} = \boldsymbol{R}^{-1}\boldsymbol{y}$ or $\boldsymbol{x} = \boldsymbol{R'}^{-1}\boldsymbol{y}$ without explicitly forming $\boldsymbol{R}^{-1}$ or $\boldsymbol{R'}^{-1}$.

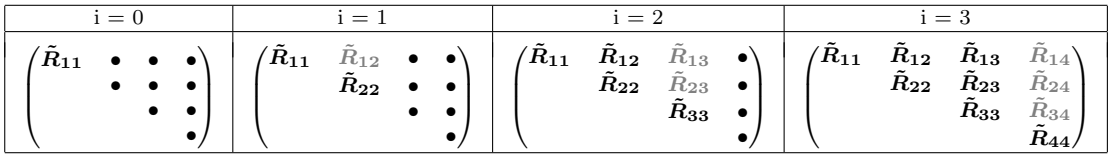


Figure 4.2: Illustration of triangular matrix inverse with overwriting using a 4×4 matrix. In each iteration, elements in diagonal blocks and rectangular block are respectively coloured in black and gray. Those elements not used in the iteration is denoted by $\bullet$.

Vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ can be generalized to $p \times k$ matrices $\boldsymbol{X}$ and $\boldsymbol{Y}$, then the problem is just k independent triangular system of equations, at a complexity of kp^2 FLOP.

Later in this thesis, a pivoted triangular system is $\boldsymbol{R}^*\boldsymbol{x} = \boldsymbol{y}$ or $\boldsymbol{R}^{*\prime}\boldsymbol{x} = \boldsymbol{y}$ is often seen, where $\boldsymbol{R}^*$ is the non-triangular factor from a QR or pivoted Cholesky factorization (and even pre-conditioning). However, this imposes no extra difficulty in equation solving. For example, in a pivoted Cholesky factorization with pre-conditioning, there is $\boldsymbol{R}^* = \boldsymbol{R}\boldsymbol{P}'\boldsymbol{J}$, where $\boldsymbol{R}$ is truely triangular. Then

$$\begin{aligned} \boldsymbol{R}^*\boldsymbol{x} = \boldsymbol{y} &\Rightarrow \boldsymbol{x} = \boldsymbol{R}^{*-1}\boldsymbol{y} = \boldsymbol{J}^{-1}\boldsymbol{P}(\boldsymbol{R}^{-1}\boldsymbol{y}), \\ \boldsymbol{R}^{*\prime}\boldsymbol{x} = \boldsymbol{y} &\Rightarrow \boldsymbol{x} = \boldsymbol{R}^{*\prime-1}\boldsymbol{y} = \boldsymbol{R}^{-1}(\boldsymbol{P}'\boldsymbol{J}^{-1}\boldsymbol{y}), \end{aligned}$$

so that the triangular structure of $\boldsymbol{R}$ can be exploited for ordinary back and forward substitution.

4.1.4 Positive-definite matrix inverse

Given a Cholesky factorization $\boldsymbol{S} = \boldsymbol{R}'\boldsymbol{R}$ for a $p \times p$ positive-definite matrix $\boldsymbol{S}$, the matrix inverse $\boldsymbol{S}^{-1} = \boldsymbol{R}^{-1}\boldsymbol{R}'^{-1}$ can be obtained by first getting triangular matrix inverse $\tilde{\boldsymbol{R}} = \boldsymbol{R}^{-1}$, then doing a matrix cross-product $\boldsymbol{S}^{-1} = \tilde{\boldsymbol{R}}\tilde{\boldsymbol{R}}'$.

A direct way to compute $\tilde{\boldsymbol{R}}$ is to solve the triangular system $\boldsymbol{R}\tilde{\boldsymbol{R}} = \boldsymbol{I}$. This appears to involve p^3 FLOP but that $\boldsymbol{I}$ is identity can make back substitution more efficient. The triangular system for $\tilde{\boldsymbol{R}}(,\ j)$ can be partitioned as

$$\begin{pmatrix} \boldsymbol{R_{11}} & \boldsymbol{R_{12}} \\ & \boldsymbol{R_{22}} \end{pmatrix} \begin{pmatrix} \boldsymbol{x_1} \\ \boldsymbol{x_2} \end{pmatrix} = \begin{pmatrix} \boldsymbol{z} \\ \boldsymbol{0} \end{pmatrix},$$

where $\boldsymbol{z}(1 : (j-1)) = \boldsymbol{0}$ and $\boldsymbol{z}(j) = 1$. Since all diagonal elements of $\boldsymbol{R}$ are positive, both $\boldsymbol{R_{11}}$ and $\boldsymbol{R_{22}}$ are invertible, thus $\tilde{\boldsymbol{R}}((j+1) : p,\ j) = \boldsymbol{x_2} = \boldsymbol{0}$ and $\tilde{\boldsymbol{R}}(1 : j,\ j) = \boldsymbol{x_1} = \boldsymbol{R_{11}^{-1}}\boldsymbol{z}$. This immediately implies that $\tilde{\boldsymbol{R}}$ is still an upper triangular matrix, and only a $j \times j$ triangular system needs be solved for $\tilde{\boldsymbol{R}}(1 : j,\ j)$. So the complexity of back substitution is reduced to $\frac{1}{3}p^3$ FLOP.

An alternative yet less obvious algorithm is practically applied in numerical linear algebra software. Partition the triangular system as

$$\begin{pmatrix} \boldsymbol{R_{11}} & \boldsymbol{R_{12}} \\ & \boldsymbol{R_{22}} \end{pmatrix} \begin{pmatrix} \tilde{\boldsymbol{R}}_{\boldsymbol{11}} & \tilde{\boldsymbol{R}}_{\boldsymbol{12}} \\ & \tilde{\boldsymbol{R}}_{\boldsymbol{22}} \end{pmatrix} = \begin{pmatrix} \boldsymbol{I_{11}} & \\ & \boldsymbol{I_{22}} \end{pmatrix},$$

then $\tilde{\boldsymbol{R}}_{\boldsymbol{11}} = \boldsymbol{R_{11}^{-1}}$, $\tilde{\boldsymbol{R}}_{\boldsymbol{22}} = \boldsymbol{R_{22}^{-1}}$ and $\tilde{\boldsymbol{R}}_{\boldsymbol{12}} = -\tilde{\boldsymbol{R}}_{\boldsymbol{11}}\boldsymbol{R_{12}}\tilde{\boldsymbol{R}}_{\boldsymbol{22}}$. Let's call $\boldsymbol{R_{11}}$ and $\boldsymbol{R_{22}}$ as leading and trailing block, respectively. There is no restriction on the size of partition, so one special case is $\boldsymbol{R_{11}} = \boldsymbol{R}(1 : (p-1),\ 1 : (p-1))$ and $\boldsymbol{R_{22}} = \boldsymbol{R}(p,\ p)$. However, for $\boldsymbol{R}(1 : (p-1),\ 1 : (p-1))$, the same kind of partition can be applied, and in this case the leading block has dimension $(p-2) \times (p-2)$. Such recursion can continue until the leading block becomes $\boldsymbol{R}(1,\ 1)$. This immediately implies an algorithm in Figure 4.3a. It has the same number of FLOP with the direct method, but it is rich in matrix-vector multiplication rather than back substitution, and is expected to be more efficient on modern computers. Croz and Higham (1992) covers both algorithms and their difference in numerical stability. Note that the algorithm can overwrite $\boldsymbol{R}$ with $\tilde{\boldsymbol{R}}$. See Figure 4.2 for an illustration.

$\tilde{\boldsymbol{R}}(1,\ 1) = 1/\boldsymbol{R}(1,\ 1)$
for $i = 1 : (p - 1)$
  $\tilde{\boldsymbol{R}}(i+1,\ i+1) = 1/\boldsymbol{R}(i+1,\ i+1)$
  $\tilde{\boldsymbol{R}}(1 : i,\ i+1) = -\tilde{\boldsymbol{R}}(i+1,\ i+1)\tilde{\boldsymbol{R}}(1 : i,\ 1 : i)\boldsymbol{R}(1 : i,\ i+1)$

(a) step 1: computing triangular matrix inverse $\tilde{\boldsymbol{R}} = \boldsymbol{R}^{-1}$.

for $i = 1 : p$
  $\boldsymbol{S}^{-1}(1 : i,\ i) = \tilde{\boldsymbol{R}}(1 : i,\ i : p)\tilde{\boldsymbol{R}}(i,\ i : p)'$

(b) step 2: computing transposed triangular matrix cross-product $\boldsymbol{S}^{-1} = \tilde{\boldsymbol{R}}\tilde{\boldsymbol{R}}'$.

Figure 4.3: Algorithm for inverting a positive-definite matrix $\boldsymbol{S}$ given its upper triangular Cholesky factor $\boldsymbol{R}$.

Transposed triangular matrix cross-product $\boldsymbol{S}^{-1} = \tilde{\boldsymbol{R}}\tilde{\boldsymbol{R}}'$ is a special type of matrix multiplication. By exploiting that $\boldsymbol{S}^{-1}$ is symmetric and $\tilde{\boldsymbol{R}}$ is upper triangular, it is not hard to obtain an algorithm in Figure 4.3b for computing its upper triangular part. The algorithm also involves $\frac{1}{3}p^3$ FLOP.

Now suppose we have a pivoted Cholesky factorization with pre-conditioning $\boldsymbol{S} = \boldsymbol{R}^{*\prime}\boldsymbol{R}^{*}$, then $\boldsymbol{R}^{*} = \boldsymbol{R}\boldsymbol{P}'\boldsymbol{J}$, where $\boldsymbol{R}$ is truely triangular. Computation of $\boldsymbol{S}^{-1}$ in this case only needs a post-processing, since $\boldsymbol{S}^{-1} = \boldsymbol{J}^{-1}\boldsymbol{P}\tilde{\boldsymbol{S}}\boldsymbol{P}'\boldsymbol{J}^{-1}$ and $\tilde{\boldsymbol{S}} = \boldsymbol{R}^{-1}\boldsymbol{R}'^{-1}$ can be first formed using methods explained above.

4.1.5 Symmetric eigen decomposition

Eigen decomposition is not just defined for a symmetric matrix, but since a symmetric $p \times p$ matrix $\boldsymbol{S}$ can be diagonalized, it gives a nice form of eigen decomposition:

$$\boldsymbol{S} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{U}',$$

where $\boldsymbol{U}$ is a $p \times p$ orthonormal matrix containing eigenvectors, and $\boldsymbol{D}$ is a diagonal matrix, whose diagonal elements are eigenvalues. This decomposition is often interpreted as an orthogonal reconstruction of $\boldsymbol{S}$, because

$$\boldsymbol{S} = \sum_{i=1}^{p} d_i \boldsymbol{u_i}\boldsymbol{u_i'},$$

where $\boldsymbol{u_i} = \boldsymbol{U}(,\ i)$ is the i^{th} eigen vector and $d_i = \boldsymbol{D}(i,\ i)$ is the i^{th} eigen value. Since $\|\boldsymbol{u_i}\| = 1$, d_i determines the magnitude of the contribution of $d_i\boldsymbol{u_i}\boldsymbol{u_i'}$. If we arrange eigenvalues so that $|d_{i+1}| \geq |d_i|$, then the contribution of these orthogonal components will be decreasing in magnitude. It is then natural to consider truncating the summation as an approximation to $\boldsymbol{S}$:

$$\boldsymbol{S_k} = \sum_{i=1}^{k} d_i \boldsymbol{u_i}\boldsymbol{u_i'} \approx \boldsymbol{S}, \quad k \leq p.$$

The larger k is, the better the approximation is. Such approximation is called a low-rank approximation, because if $\boldsymbol{S}$ has a full rank of p, $\boldsymbol{S_k}$ only has a rank of k.

The above eigen decomposition implies that a matrix "root" of $\boldsymbol{S}$ can be computed as $\boldsymbol{S}^{\frac{1}{2}} = \boldsymbol{U}\boldsymbol{D}^{\frac{1}{2}}\boldsymbol{U}'$, where $\boldsymbol{D}^{\frac{1}{2}}$ is a diagonal matrix with $\boldsymbol{D}^{\frac{1}{2}}(i,\ i) = \sqrt{d_i}$, so that $(\boldsymbol{S}^{\frac{1}{2}})^{\boldsymbol{2}} = \boldsymbol{S}^{\frac{1}{2}}\boldsymbol{S}^{\frac{1}{2}} = \boldsymbol{S}$. When $\boldsymbol{S}$ is positive-definite or positive-semidefinite, its root is still a real matrix. However, in this thesis, $\boldsymbol{S}^{\frac{1}{2}}$ will be taken as the upper triangular Cholesky factor of $\boldsymbol{S}$, so that there is instead $(\boldsymbol{S}^{\frac{1}{2}})'\boldsymbol{S}^{\frac{1}{2}} = \boldsymbol{S}$.

Unlike any other matrix computation in this preliminary section, eigen decomposition is the only one with no exact FLOP count, because its practical computation is based on iterative methods. Precisely it is a two-stage task:

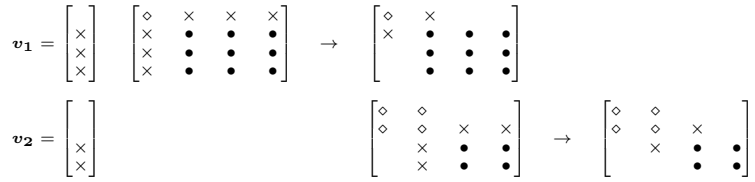


Figure 4.4: Illustration of Householder tridiagonalization using a 4×4 matrix. See Figure 4.1 for meanings of place holders $\bullet$, $\times$ and $\diamond$.

Table 4.1: A brief summary of the three matrix factorizations elaborated in §4.1. "$\sim$ upper triangular" means "equivalent to upper triangular"; that is, $\boldsymbol{R}^*$ is upper triangular upon a column permutation.

	QR	Cholesky	symmetric eigen
math form	$\boldsymbol{X} = \boldsymbol{Q}\boldsymbol{R}^*$	$\boldsymbol{S} = \boldsymbol{R}^{*\prime}\boldsymbol{R}^*$	$\boldsymbol{S} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{U}'$
input	$\boldsymbol{X}$ (general)	$\boldsymbol{S}$ (positive-semidefinite)	$\boldsymbol{S}$ (symmetric)
dimension	$n \times p$ $(n \geq p)$	$p \times p$	$p \times p$
output 1	$\boldsymbol{Q}$ ("factored form")	$\boldsymbol{R}^*$ ($\sim$ upper triangular)	$\boldsymbol{U}$ (orthogonal)
output 2	$\boldsymbol{R}^*$ ($\sim$ upper triangular)		$\boldsymbol{D}$ (diagonal)
FLOP	$2np^2 - \frac{2}{3}p^3$	$\frac{1}{3}p^3$	$> \frac{4}{3}p^3$

1. $\boldsymbol{S}$ is first tri-diagonalized by sequential Householder transformations from both left and right. This is similar to what happens in a QR factorization and Figure 4.4 is an illustration. Golub and Loan (2013, §8.3.1) gives a detailed algorithm for this, involving $\frac{4}{3}p^3$ FLOP;
2. QR iteration is then applied on the tri-diagonal matrix to find eigenvalues. It will be too lengthy if I cover this topic here, but Golub and Loan (2013, §8.3) gives substantial details for anyone with interest. Each iteration has $O(p^2)$ FLOP.

4.1.6 Summary

This section is a consolidation of what has been covered so far. The three types of matrix factorizations are probably the most confusing to you, so I specially list them in Table 4.1. There is no implication that one factorization is more useful than the others; they are all essential in linear algebra. QR factorization is distinguished from the other two as it has almost no requirement on the input matrix. Symmetric eigen decomposition needs a symmetric matrix. Cholesky factorization has the strongest input requirement: all eigen values of the input symmetric matrix must be non-negative.

In practice, matrix factorizations can be used in various ways for different purposes. Here I will provide a few examples.

Solving a linear system of equations

They may be used for solving a linear equation $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$. QR factorization can be used to solve a very general system.

- For a full-rank system (exclusively when $\boldsymbol{A}$ is square and has full rank) it is an alternative to Gaussian elimination or LU factorization;
- For an over-determined system (for example, when $\boldsymbol{A}$ has more rows than columns) it yields the least squares solution;
- For an under-determined system (for example, when $\boldsymbol{A}$ has more columns than rows) it constrains free parameters to zero and estimate a truncated full-rank system. This is also the way

that `R` function `lm` finds a least squares solution for rank-deficient system (i.e., when A has more rows than columns but it does not have full column rank).

Cholesky factorization can also be used for equation solving, but it only works for a positive-definite system, where $\boldsymbol{A}$ is square, symmetric and full-rank. $\boldsymbol{A}$ can be relaxed to be positive-semidefinite, in which case the system is essentially over-determined. Using Cholesky factorization with pivoting, numerical rank of the system can be estimated and a truncation can be performed, constraining free parameters to zeros hence yielding a least squares solution. Eigen decomposition can be used as an alternative to Cholesky factorization for solving this type of system, but in practice this is rarely done as it is more computationally costly.

Note that if the RHS of the system is zero, we get a homogenous system. The solution gives the *null space* of $\boldsymbol{A}$. We will from time to time see a homogenous system. For example, find the eigenvectors for $\boldsymbol{A}$ associated with an eigenvalue λ. Mathematically eigenvectors are solutions to $(\boldsymbol{A} - \lambda \boldsymbol{I})\boldsymbol{x} = \boldsymbol{0}$. Of course, such is only used for educational purpose like in a textbook of undergraduate linear algebra. In practice, eigenvalues and eigenvectors are both unknown and can be found together using eigen decomposition (note: that I only introduced symmetric eigen decomposition does not mean eigen decomposition is not defined for general matrix; eigenvalues and eigenvalues are the most important character for a square matrix).

GAM estimation involves solving a penalized least squares problem (1.4). An in fact, it is just a least squares problem and hence an equation solving problem. Consider a pivoted Cholesky factorization $\boldsymbol{S}_{\boldsymbol{\lambda}} = \boldsymbol{E}_{\boldsymbol{\lambda}}^{*\prime}\boldsymbol{E}_{\boldsymbol{\lambda}}^{*}$ (the Cholesky factor also depends on $\boldsymbol{\lambda}$), it is straightforward to rewrite the penalized least squares object as

$$\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}} = \arg\min_{\boldsymbol{\beta}} \left\| \begin{pmatrix} \boldsymbol{W}^{\frac{1}{2}}\boldsymbol{y} \\ \boldsymbol{0} \end{pmatrix} - \begin{pmatrix} \boldsymbol{W}^{\frac{1}{2}}\boldsymbol{X} \\ \boldsymbol{E}_{\boldsymbol{\lambda}} \end{pmatrix} \boldsymbol{\beta} \right\|^2, \tag{4.1}$$

which is a least squares problem with an augmented model matrix and an augmented response vector.

Setting linear constraints on parameters

A linear system is also often used to specify linear constraints on parameters. This is mostly seen in constrained optimization problems. However, here in GAM computations I can give you a concrete example: centring constraint on a spline.

Since the individual $\boldsymbol{f_i}$ in an additive model is only estimable to within an intercept term, identifiability constraints need be applied. As discussed in Wood et al. (2013), a *sum-to-zero constraint* has the advantage of leading to narrow confidence intervals on the constrained f_i. For example, for f_1 above this is $\sum_{i=1}^{n} f_1(x_i) = 0$ and for f_2 this is $\sum_{i=1}^{n} f_2(z_{i1},\ z_{i2}) = 0$. Such constraint can also be written as a matrix equation $\boldsymbol{C}_{\boldsymbol{i}}'\boldsymbol{X_i} = 0$, where $\boldsymbol{C_i} = \boldsymbol{X_i'1}$ is a vector containing column-wise sum of $\boldsymbol{X_i}$.

It is easy to reparametrize $\boldsymbol{\beta_i}$ to incorporate the constraint directly into $\boldsymbol{X_i}$ and $\boldsymbol{S_i}$. The constraint implies that $\boldsymbol{\beta_i}$ lies in the null space of $\boldsymbol{C_i'}$. Then we can solve the homogenous system $\boldsymbol{C_i'X_i} = 0$ for the null space basis $\boldsymbol{B_i}$. Clearly there exists some unconstrained parameters $\tilde{\boldsymbol{\beta}}_{\boldsymbol{i}}$ such that $\boldsymbol{\beta_i} = \boldsymbol{B_i}$. So instead of estimating $\boldsymbol{\beta_i}$ we can estimate $\tilde{\boldsymbol{\beta}}_{\boldsymbol{i}}$. This is essentially a reparametrization so that we have a new design matrix $\boldsymbol{X_iB_i}$ and new coefficient vector $\tilde{\boldsymbol{\beta}}_{\boldsymbol{i}}$ for spline $\boldsymbol{f_i}$.

Note that the QR factorization provides another way for finding $\boldsymbol{B_i}$ than equation solving. As metioned in §4.1.1, following a thin QR factorization $\boldsymbol{A} = \boldsymbol{QR}$, $\boldsymbol{Q}^{\perp}$ is an orthonormal basis for the null space of $\boldsymbol{A'}$. Therefore, to find an orthonormal basis for $\boldsymbol{A}$ we just need to apply QR factorization to $\boldsymbol{A'}$. So, following a thin QR factorization $\boldsymbol{C_i} = \boldsymbol{Q_iR_i}$, spline $\boldsymbol{f_i}$ can be reparametrized using $\boldsymbol{B_i} = \boldsymbol{Q_i}^{\perp}$, ending up with a new design matrix $\boldsymbol{X_iQ_i}^{\perp}$. As has been explained in §4.1.1, matrix multiplication with $\boldsymbol{Q_i}^{\perp}$ is effectively a set of Householder transformations. In particular, as $\boldsymbol{C_i}$ is a

single-column matrix, there will be just one Householder transformation here. The new design matrix is one column less, and the new penalty matrix will lose a row and a column. This reparametrization is said to "absorb centring constraint into $\boldsymbol{f_i}$".

Variable transformation

Absorbing centring constrain is an example of variable transformation using QR factorization. Now I will give other examples for Cholesky factorization and eigen decomposition.

In §2.2.2 I have defined the standardization for an AR(1) process. Broadly speaking this provides a way to make transformations between variables with a general covariance matrix and variables with an identity covariance matrix. That is, we can sample from a Gaussian distribution with a general covariance matrix by transforming i.i.d. Gaussian samples, and vice versa.

- Suppose we want to sample $\boldsymbol{x} \sim N(\boldsymbol{0},\ \boldsymbol{V}\phi)$. Consider a Cholesky factorization $\boldsymbol{V} = \boldsymbol{L}\boldsymbol{L'}$ where $\boldsymbol{L}$ is the lower triangular Cholesky factor. x can be obtained by first sampling $\boldsymbol{z} \sim N(\boldsymbol{0},\ \boldsymbol{I}\phi)$ then setting $\boldsymbol{x} = \boldsymbol{L}\boldsymbol{z}$. (The reverse statement is: $\boldsymbol{x}$ can be standardized via $\boldsymbol{L}^{-1}\boldsymbol{x}$.)
- Suppose we want to sample $\boldsymbol{x} \sim N(\boldsymbol{0},\ \boldsymbol{W}^{-1}\phi)$. Consider a Cholesky factorization $\boldsymbol{W} = \boldsymbol{U'}\boldsymbol{U}$ where $\boldsymbol{U}$ is the upper triangular Cholesky factor. x can be obtained by first sampling $\boldsymbol{z} \sim N(\boldsymbol{0},\ \boldsymbol{I}\phi)$ then setting $\boldsymbol{x} = \boldsymbol{U}^{-1}\boldsymbol{z}$. (The reverse statement is: $\boldsymbol{x}$ can be standardized via $\boldsymbol{U}^{-1}\boldsymbol{x}$.)

Eigen decomposition can be used for the same task. In the first case, decompose the correlation matrix $\boldsymbol{V} = \boldsymbol{E}\boldsymbol{D}\boldsymbol{E'}$, then we have $\boldsymbol{L} = \boldsymbol{E}\boldsymbol{D}^{\frac{1}{2}}$. In the second case, decompose the inverse correlation matrix $\boldsymbol{W} = \boldsymbol{E}\boldsymbol{D}\boldsymbol{E'}$, then we have $\boldsymbol{U} = \boldsymbol{D}^{\frac{1}{2}}\boldsymbol{E'}$. (Note that $\boldsymbol{L}$ and $\boldsymbol{U}$ are not triangular matrices in this method.)

Computing determinant of a matrix

Matrix factorizations provide a way to compute determinant of a square matrix.

- For a general square matrix, LU factorization is used: $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{U}$, where $\boldsymbol{L}$ is a unit lower triangular matrix (a lower triangular matrix will all ones on its main diagonal) and $\boldsymbol{U}$ is an upper triangular matrix. Then, $|\boldsymbol{A}| = |\boldsymbol{L}||\boldsymbol{U}| = |\boldsymbol{U}|$. (I haven't introduced LU factorization in the previous sections, so there is no need to know anything further than this).
- For a general symmetric matrix, symmetric eigen factorization can be used: $\boldsymbol{A} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{U'}$, then $|\boldsymbol{A}| = |\boldsymbol{U}||\boldsymbol{D}||\boldsymbol{U'}| = |\boldsymbol{U}|^2|\boldsymbol{D}| = |\boldsymbol{D}| = \prod_i \boldsymbol{D}(i,\ i)$.
- For a positive-definite matrix, Cholesky factorization can be used: $\boldsymbol{A} = \boldsymbol{R'}\boldsymbol{R}$, then $|\boldsymbol{A}| = |\boldsymbol{R}||\boldsymbol{R'}| = |\boldsymbol{R}|^2 = \prod_i \boldsymbol{R}(i,\ i)^2$.

GAM estimation also involves computation of determinant (or actually log-determinant). If you still remember (check (1.9)), there are three determinant terms in the REML score $\mathcal{V}_r$ (revisit §1.1.6 if you need to): $\log|\boldsymbol{S}_{\boldsymbol{\lambda}}|_+$, $\log|\boldsymbol{H}_{\boldsymbol{\lambda}}|$ and $\log|\boldsymbol{W}|$. The last determinant is mostly trivial. Very often $\boldsymbol{W}$ is just a diagonal matrix of weights w_i, so the determinant is just $|\boldsymbol{W}| = \prod_i w_i$ or $\log|\boldsymbol{W}| = \sum_i \log(w_i)$. In a slightly more complicated case $\boldsymbol{W}$ is symmetric tri-diagonal coming from AR(1) model errors, and in §2.2.2 I have already shown what this determinant is. In the forthcoming sections of this Chapter, you will see how $\log|\boldsymbol{S}_{\boldsymbol{\lambda}}|_+$ and $\log|\boldsymbol{H}_{\boldsymbol{\lambda}}|$ are computed using matrix factorizations.

Achieving computational efficiency

Generally if you do some similar matrix computations in a loop where one matrix is loop invariant, there might be a change to speed up the loop by an appropriate initial factorization of this matrix.

That matrix factorization helps achieve computational efficiency will be seen a few occasions in the rest of this Chapter, like the model matrix reduction in §4.2 and the initial reparametrization for REML estimation in §4.3.1. Here I will provide another example.

Let $\boldsymbol{A}$ be a $p \times p$ real symmetric matrix. How would you compute a matrix power series $\sum_{i=1}^{m} \boldsymbol{A^i}$? The obvious yet the worst idea is to do the summation via a loop, computing $\boldsymbol{A^i}$ by $(i-1)$ matrix multiplication $\prod_{j=1}^{i} \boldsymbol{A_j}$ in the i^{th} iteration. This involves $O(m^2p^3)$ FLOP. A better arrangement is to accumulate $\boldsymbol{B_i} = \prod_{j=1}^{i} \boldsymbol{A_j}$ along the loop so that only one matrix multiplication $\boldsymbol{B_{i+1}} = \boldsymbol{B_i A}$ is needed in each iteration. This reduces the computational complexity to $O(mp^3)$.

Now consider a symmetric eigen decomposition $\boldsymbol{A} = \boldsymbol{UDU'}$, then we see $\sum_{i=1}^{m} \boldsymbol{A^i} = \sum_{i=1}^{m} (\boldsymbol{UDU'}) = \boldsymbol{U}(\sum_{i=1}^{m} \boldsymbol{D^i})\boldsymbol{U'}$. Note that $\boldsymbol{D}$ is a diagonal matrix with diagonal elements d_1, d_2, ..., d_p, so that its power series is just

$$\sum_{i=1}^{m} \begin{pmatrix} d_1 & & & \\ & d_2 & & \\ & & \ddots & \\ & & & d_p \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^{m} d_1 & & & \\ & \sum_{i=1}^{m} d_2 & & \\ & & \ddots & \\ & & & \sum_{i=1}^{m} d_p \end{pmatrix},$$

where the j^{th} element can be analytically computed by $\sum_{i=1}^{m} d_j = \frac{d_j(1-d_j^m)}{1-d_j}$ without a loop. So the computational complexity of this method is really just the $O(p^3)$ FLOP for the initial decomposition; the summatition loop has no costs at all!

Conclusion

I have presented rich contents here on matrix computations. While they are lengthy, they are right on the theme of this Chapter (and this thesis!). You will shortly see many matrix operations and if these operations are new to you, you can easily be buried in details. In this situation, do remind yourself of the aim the matrix computations summarized in the headings for the above paragraphs.

4.2 Model matrix reduction for additive models

The core of estimating an additive model is just solving a penalized least squares (1.4) or (4.1). And if the smoothing parameters $\boldsymbol{\lambda}$ are unknown, the problem is iteratively solved many times until the best $\boldsymbol{\lambda}$ is found that minimizes, say, the REML score $\mathcal{V}_r$. The most obvious way to do this iterative search is a gird search. However, I have already stated in S 2.2.2 that a grid search is much too inefficient. If the parameter space has high dimension, this simple method would be prohibitively costly (as the number of grid points required grows geometrically with dimension; this is known as the "curse of dimensionality"). Genearlly a guided search is performed. For example, the golden-section search in §2.2.3 is just a simple guided search for 1D optimization that converges fast (enough). Minimization of the REML score $\mathcal{V}_r$ is a high dimensional optimization (as there can be many splines and smoothing parameters), and Newton-Raphson algorithm is a guided search that converges fast (enough).

More details on Newton-Raphson algorithm will be provided in the next section (§4.3). For the moment, let us focus on that fact that (4.1) needs be solved many times in a loop. Remember the "Achieving computational efficiency" headings in the previous section? Obviously $\boldsymbol{X}$ is loop-invariant and there might be an opportunity to apply matrix factorization here. For additive models with known $\boldsymbol{W}$, $\boldsymbol{W}^{\frac{1}{2}}\boldsymbol{X}$ will be loop-invariant. Consider a pivoted QR factorization (see §4.1.1 if you need a revision) $\boldsymbol{W}^{\frac{1}{2}}\boldsymbol{X} = \boldsymbol{QR}^*$, then $D_p(\boldsymbol{\beta})$ in (1.5) can be rewritten as

$$D_p(\boldsymbol{\beta},\ \boldsymbol{\lambda}) = \left\|\boldsymbol{f} - \boldsymbol{R}^*\boldsymbol{\beta}\right\|^2 + \boldsymbol{\beta'}\boldsymbol{S_\lambda}\boldsymbol{\beta} + \left\|\boldsymbol{W}^{\frac{1}{2}}\boldsymbol{y}\right\|^2 - \|\boldsymbol{f}\|^2, \tag{4.2}$$

where $\boldsymbol{f} = \boldsymbol{Q'W}^{\frac{1}{2}}\boldsymbol{X}$. Note that $\left\|\boldsymbol{W}^{\frac{1}{2}}\boldsymbol{y}\right\|^2 - \|\boldsymbol{f}\|^2$ is also loop-invariant; so what really needs be solved in a loop is a reduced penalized least squares problem

$$\left\|\boldsymbol{f} - \boldsymbol{R}^*\boldsymbol{\beta}\right\|^2 + \boldsymbol{\beta'}\boldsymbol{S}_{\boldsymbol{\lambda}}\boldsymbol{\beta}.$$

Although I haven't told you what the computational costs associated with a penalized least squares problem is, you should have observed that matrices in the reduced problem have smaller size than those in the original version (1.4) or (4.1), hence been able to infer that solving the reduced version in a loop is much cheaper.

In fact, for large datasets (i.e., large n) this is double gains. The model matrix $\boldsymbol{X}$ has large number of rows which can take huge amount of RAM for storage. For example, a 300000×10000 matrix takes 22.35 GB. The reduced penalized least squares objective eliminates the need to access it every iteration hence the loop will be more memory efficient.

Such initial QR factorization for the model matrix $\boldsymbol{X}$ to obtain $\boldsymbol{f}$ and $\boldsymbol{R}^*$ prior to $\mathcal{V}_r$ minimization, is termed *model matrix reduction*, or precisely the *QR reduction* variant in this thesis. There is an alternative variant through matrix cross-product $\boldsymbol{X'WX}$ and a subsequent Cholesky factorization, termed *pseudo QR reduction*. This section will give computational details for both variants.

You probably wonder why this initial QR factorization worths writing a section, given that QR factorization has been well explained in §4.1.1. Well, the issue is that $\boldsymbol{X}$ is so large. Even if it won't be accessed in the REML estimation loop, a standard QR factorization still requires this large matrix to be in memory in the first place. Holding this matrix in RAM is challenging or even impossible on most computational devices. Particularly, there is no upper bound for n, which means there is no upper bound for memory usage in this standard way. The sections on model matrix reduction here aim to demonstrate an implementation where memory requirement is a constant that is independent of n. It will also turn out that such implementation can exploit parallel computing for speed.

To simplify notation in the following demonstration, I will assume that the model matrix and the response vector have absorbed weights: $\boldsymbol{X} \leftarrow \boldsymbol{W}^{\frac{1}{2}}\boldsymbol{X}$, $\boldsymbol{y} \leftarrow \boldsymbol{W}^{\frac{1}{2}}\boldsymbol{y}$.

4.2.1 QR reduction

In fact, the solution to avoid forming $\boldsymbol{X}$ as a whole has been established, by using a QR updating algorithm for iterative update of a QR factorization(Sven and Craig, 2008, §2.3.2).

The QR updating problem is an example of many "online" problems. Here a simple example to help understand what an "online" problem is. Suppose we want to computed and report sample variance of a dataset where a new datum arrives every second. Assume that the dataset initially has a single datum at second *1*, then it will contain t data by second t and we want to report their sample variance v_t. A poor man's approach would be using formula $m_t = \frac{1}{t}\sum_{i=1}^{t} x_i$, $v_t = \sum_{i=1}^{t}(x_i - m_i)^2$, which scans every record up to time t thus involves $O(t)$ computation complexity. This implies that computation is increasingly time-consuming as time goes by, and eventually we won't be able to output v_t as fast as x_t arrives. Do we have to scan the data vector right from the beginning every second? Is there a way to simply update v_{t-1} to get v_t? Yes, there is. Starting with $m_1 = x_1$ and $s_1 = 0$, the Welford's online algorithm(Welford, 1962) suggests $m_t = m_{t-1} + (x_t - x_{t-1})\ /t$, $s_t = s_{t-1} + (x_t - m_{t-1})(x_t - m_t)$, $v_t = s_t\ /\ t$ which updates the sample mean m_t and sample variance v_t at $O(1)$ computational complexity for any $t \geq 2$. Such algorithm is called "online" because it does not visit or store old records: we may discard x_{t-1} as soon as m_t and v_t are obtained.

Now let us get back to the QR factorization problem. Partition the large model matrix $\boldsymbol{X}$ and

response vector $\boldsymbol{y}$ into row chunks:

$$\boldsymbol{X} = \begin{pmatrix} \boldsymbol{X_1} \\ \boldsymbol{X_2} \\ \vdots \\ \boldsymbol{X_i} \\ \vdots \end{pmatrix}, \qquad \boldsymbol{y} = \begin{pmatrix} \boldsymbol{y_1} \\ \boldsymbol{y_2} \\ \vdots \\ \boldsymbol{y_i} \\ \vdots \end{pmatrix},$$

and imagine those chunks arrive by time. There are no restrictions on the size of each chunk, but to ease practical implementation let all chunks have equal row chunk size c (although the final chunk may have fewer rows if c does not divide n). Denote $\boldsymbol{X}^{[i]}$ and $\boldsymbol{y}^{[i]}$ as the previous i row chunks of $\boldsymbol{X}$ and $\boldsymbol{y}$, then there are the following update formulas.

1. Given a pivoted QR factorization $\boldsymbol{X}^{[i]} = \boldsymbol{Q_i R_i^*}$, the pivoted QR factorization for $\boldsymbol{X}^{[i+1]}$ is only up to a factorization of $\begin{pmatrix} \boldsymbol{R_i^*} \\ \boldsymbol{X_{i+1}} \end{pmatrix}$, since

   $$X^{[i+1]} = \begin{pmatrix} X^{[i]} \\ X_{i+1} \end{pmatrix} = \begin{pmatrix} Q_i R_i^* \\ X_{i+1} \end{pmatrix} = \begin{pmatrix} Q_i & \\ & I \end{pmatrix} \begin{pmatrix} \boldsymbol{R_i^*} \\ \boldsymbol{X_i} \end{pmatrix} = \begin{pmatrix} Q_i & \\ & I \end{pmatrix} \tilde{\boldsymbol{Q}}_{i+1} \boldsymbol{R_{i+1}^*}.$$

   So $\boldsymbol{Q_{i+1}} = \begin{pmatrix} \boldsymbol{Q_i} & \\ & \boldsymbol{I} \end{pmatrix} \tilde{\boldsymbol{Q}}_{i+1}$ and $\boldsymbol{R_{i+1}^*}$ are 'Q' and 'R' factors of $\boldsymbol{X}^{[i+1]}$.

2. Given $\boldsymbol{f_i} = \boldsymbol{Q_i'} \boldsymbol{y}^{[i]}$, $\boldsymbol{f_{i+1}} = \boldsymbol{Q_{i+1}'} \boldsymbol{y}^{[i+1]}$ can be obtained by an update

   $$\boldsymbol{f} = \boldsymbol{Q_{i+1}'} \boldsymbol{y}^{[i+1]} = \tilde{\boldsymbol{Q}}_{i+1}' \begin{pmatrix} \boldsymbol{Q_i'} & \\ & \boldsymbol{I} \end{pmatrix} \begin{pmatrix} \boldsymbol{y}^{[i]} \\ \boldsymbol{y_i} \end{pmatrix} = \tilde{\boldsymbol{Q}}_{i+1}' \begin{pmatrix} \boldsymbol{f_i} \\ \boldsymbol{y_i} \end{pmatrix}.$$

   In other words, there is no realistic need to form $\boldsymbol{Q_{i+1}}$.

The above is an "online" algorithm, because

1. $\boldsymbol{R_{i+1}^*}$ and $\boldsymbol{f_{i+1}}$ are updated from $\boldsymbol{X_i}$, $\boldsymbol{R_i^*}$ and $\boldsymbol{f_i}$ rather than computed from $\boldsymbol{X}^{[i+1]}$ and $\boldsymbol{y}^{[i+1]}$;
2. as soon as $\boldsymbol{R_{i+1}}$ and $\boldsymbol{f_{i+1}^*}$ are obtained $\boldsymbol{X_i}$ can be discarded.

As a result, only a single row chunk of $\boldsymbol{X}$ and $\boldsymbol{y}$ need be in RAM for this "online updating", eliminating the storage requirement for the whole $\boldsymbol{X}$. For later reference, let us call the basic implementation where the whole $\boldsymbol{X}$ is formed the "offline" implementation.

Figure 4.5 gives a practical implementation of the "online" algorithm. In each iteration, only a $\tilde{c} \times p$ row chunk of $\boldsymbol{X}$ is formed, and only $O(cp + p^2)$ storage is needed in RAM for the QR factorization.

It is also easy to count that the algorithm involves $\left(2np^2 - \frac{2}{3}p^3 + 2(\frac{n}{c} - 1)p^3\right)$ FLOP. Compared with the $(2np^2 - \frac{2}{3}p^3)$ FLOP in the "offline" version, the algorithm has a $2(\frac{n}{c} - 1)p^3$ overhead, which is approximately $\frac{p}{c}$ of the $(2np^2 - \frac{2}{3}p^3)$ useful work when $n \gg p$. To keep this overhead low, it makes sense to choose c as large as possible. But under a memory constraint $cp < M$, we can only set $c = \frac{M}{p}$ at best, thus the percentage of overhead becomes p^2/M. This implies that on a machine with a certain amount of RAM, the overhead is negligible for small p, but has a quadratic growth for increasingly larger p. This is a major drawback of the algorithm.

4.2.2 Pseudo QR reduction

I will now introduce the other variant for model matrix reduction: pseudo QR reduction. It is motivated by the link between Cholesky factorization and QR factorization, as described in §4.1.2.

initialize $\boldsymbol{R}^*$ as a $0 \times p$ matrix and $\boldsymbol{f}$ as a length-0 vector
$\boldsymbol{R}^* = \text{zeros}(0,\ p)$
$\boldsymbol{f} = \text{zeros}(0)$
$i = 1$
while $(i \leq n)$
  determine the row chunk size for the current iteration
  $\tilde{c} = \min\{c,\ n - i + 1\}$
  form a row chunk of $\boldsymbol{X}$ and $\boldsymbol{y}$
  $\tilde{\boldsymbol{X}} = \boldsymbol{X}(i : (i + \tilde{c} - 1),\ )$
  $\tilde{\boldsymbol{y}} = \boldsymbol{y}(i : (i + \tilde{c} - 1))$
  QR factorization and orthonormal transformation
  $\begin{pmatrix} \boldsymbol{R}^* \\ \tilde{\boldsymbol{X}} \end{pmatrix} = \tilde{\boldsymbol{Q}}\tilde{\boldsymbol{R}}$
  $\tilde{\boldsymbol{f}} = \tilde{\boldsymbol{Q}}'\begin{pmatrix} \boldsymbol{f} \\ \tilde{\boldsymbol{y}} \end{pmatrix}$
  update $\boldsymbol{R}^*$ and $\boldsymbol{f}$
  $\boldsymbol{R}^* \leftarrow \tilde{\boldsymbol{R}}$
  $\boldsymbol{f} \leftarrow \tilde{\boldsymbol{f}}$
  move to next row chunk
  $i \mathrel{+}= \tilde{c}$

Figure 4.5: "Online" QR reduction with a row chunk size c. By forming a row chunk of $\boldsymbol{X}$ per iteration, the memory footprint is bounded by $O(cp + p^2)$. The QR factorization in each iteration involves $(2\tilde{c}p^2 + \frac{4}{3}p^3)$ FLOP, and the algorithm involves $(2np^2 - \frac{2}{3}p^3 + 2(\frac{n}{c} - 1)p^3)$ FLOP.

By first forming a matrix cross-product $\boldsymbol{X'X}$, a subsequent pivoted Cholesky factorization (with pre-conditioning for computational stability) would yield $\boldsymbol{R}^*$. However, $\boldsymbol{f}$ is not available without a 'Q' factor, but the QR factorization implies $\boldsymbol{Q} = \boldsymbol{X}\boldsymbol{R}^{*-1}$, hence we have $\boldsymbol{f} = \boldsymbol{Q'y} = \boldsymbol{R}^{*\prime-1}(\boldsymbol{X'y})$, which can be computed by solving a triangular system of linear equations (see §4.1.3 if you need a revision). Note that when $\boldsymbol{X}$ is rank-deficient, $\boldsymbol{R}^*$ is not invertible. Computation of $\boldsymbol{f}$ then proceeds as follows. Recall from §4.1.2 that we have $\boldsymbol{R}^* = \boldsymbol{RP'J}$ where $\boldsymbol{R}$ is an upper triangular matrix. Let r be the rank of $\boldsymbol{R}$, then we first initialize $\boldsymbol{f}$ as a length-p vector of zeros, then set $\boldsymbol{f}(1:r) = \boldsymbol{R}(1:r,\ 1:r)^{-1}\big((\boldsymbol{P'J}^{-1}\boldsymbol{X'y})(1:r)\big)$.

A big advantage of pseudo QR reduction over QR reduction is its efficiency. It involves np^2 FLOP for the cross-product, $\frac{1}{3}p^3$ FLOP for the Cholesky factorization and p^2 FLOP for solving the triangular system, hence a total of $(np^2 + \frac{1}{3}p^3 + p^2)$ FLOP, approximately half of the $(2np^2 - \frac{2}{3}p^3)$ FLOP with QR reduction when $n \gg p$.

Pseudo QR reduction also has an "online" implementation. After row chunking of $\boldsymbol{X}$ and $\boldsymbol{y}$, there are $\boldsymbol{X'X} = \sum_i \boldsymbol{X}_i'\boldsymbol{X}_i$ and $\boldsymbol{X'y} = \sum_i \boldsymbol{X}_i'\boldsymbol{y}_i$, that is, we form one row chunk in each iteration and accumulate $\boldsymbol{X'X}$ and $\boldsymbol{X'y}$. Figure 4.6 is a sketch of its practical implementation. A key message is that the algorithm has no chunking overhead compared with the non-iterative version.

4.2.3 Parallel computing

The "online" algorithms for both reduction methods remove memory bottleneck, but do not improve the fundamental $O(np^2)$ computational complexity. So it is expected that as n grows, computation becomes more time consuming.

A practical idea to speed up the computation is *parallel computing*. For example, if a loop runs 100 iterations, we may do the first 50 iterations on one processor and the other 50 iterations on a second processor. Provided that the latter 50 iterations do not rely on the processing results of the former 50 iterations, we can execute the two parts at the same time (or in parallel), so that while the total amount of work remains unchanged, the realistic execution time may be reduced (ideally) by half.

initialize a $p \times p$ matrix of zeros and a length-p vector of zeros
$\boldsymbol{A} = \text{zeros}(p,\ p)$
$\boldsymbol{z} = \text{zeros}(p)$
$i = 1$
while $(i \leq n)$
  determine the row chunk size for the current iteration
  $\tilde{c} = \min\{c,\ n - i + 1\}$
  form a row chunk of $\boldsymbol{X}$ and $\boldsymbol{y}$
  $\tilde{\boldsymbol{X}} = \boldsymbol{X}(i : (i + \tilde{c} - 1),\ )$
  $\tilde{\boldsymbol{y}} = \ \boldsymbol{y}(i : (i + \tilde{c} - 1))$
  accumulate matrix-matrix cross-product and matrix-vector cross-product
  $\boldsymbol{A}$ += $\tilde{\boldsymbol{X}}'\tilde{\boldsymbol{X}}$
  $\boldsymbol{z}$ += $\tilde{\boldsymbol{X}}'\tilde{\boldsymbol{y}}$
  move to next row chunk
  i += $\tilde{c}$
Cholesky factorization and solving a triangular system
$\boldsymbol{A} = \boldsymbol{R}^{*\prime}\boldsymbol{R}^{*}$
$\boldsymbol{f} = \boldsymbol{R}^{*\prime -1}\boldsymbol{z}$

Figure 4.6: "Online" pseudo QR reduction with a row chunk size c. By forming a row chunk of $\boldsymbol{X}$ per iteration, the memory footprint is bounded by $O(cp)$. The cross-product in each iteration involves $\tilde{c}p^2$ FLOP, the final Cholesky factorization involves $\frac{1}{3}p^3$ FLOP and solving the triangular system involves p^2 FLOP. The algorithm involves a total of $(np^2 + \frac{1}{3}p^3 + p^2)$ FLOP.

For later reference, computation performed strictly on a single processor is called *serial computing*.

It is probably not obvious how algorithms in Figures 4.5 and 4.6 can be made parallel, as there is loop carried dependence: the $(i+1)^{\text{th}}$ iteration relies on the execution result of the i^{th} iteration. However, both algorithms can be adapted to break such dependency. Suppose there are $m > 1$ processors. We split $\boldsymbol{X}$ and $\boldsymbol{y}$ into m chunks:

$$\begin{pmatrix} \boldsymbol{X_1} \\ \boldsymbol{X_2} \\ \vdots \\ \boldsymbol{X_m} \end{pmatrix}, \qquad \boldsymbol{y} = \begin{pmatrix} \boldsymbol{y_1} \\ \boldsymbol{y_2} \\ \vdots \\ \boldsymbol{y_m}. \end{pmatrix}.$$

For QR reduction method, let the j^{th} processor perform $\boldsymbol{X_j} = \boldsymbol{Q_j}\boldsymbol{R_j}*$ and $\boldsymbol{f_j} = \boldsymbol{Q}_j'\boldsymbol{y_j}$ using the "online" algorithm in Figure 4.5, then the following identity implies that processing results $\boldsymbol{R_j}$'s from all processors can be stacked for a final QR factorization to produce the 'Q' and 'R' factors associated with the complete $\boldsymbol{X}$.

$$\begin{pmatrix} X_1 \\ X_2 \\ \vdots \\ X_m \end{pmatrix} = \begin{pmatrix} Q_1 & & & \\ & Q_2 & & \\ & & \ddots & \\ & & & Q_m \end{pmatrix} \begin{pmatrix} \boldsymbol{R}_1^* \\ \boldsymbol{R}_2^* \\ \vdots \\ \boldsymbol{R}_m^* \end{pmatrix} = \begin{pmatrix} Q_1 & & & \\ & Q_2 & & \\ & & \ddots & \\ & & & Q_m \end{pmatrix} \boldsymbol{Q}\boldsymbol{R}^*.$$

It also follows that

$$\boldsymbol{f} = \boldsymbol{Q}' \begin{pmatrix} \boldsymbol{Q_1} & & & \\ & \boldsymbol{Q_2} & & \\ & & \ddots & \\ & & & \boldsymbol{Q_m} \end{pmatrix} \begin{pmatrix} \boldsymbol{y_1} \\ \boldsymbol{y_2} \\ \vdots \\ \boldsymbol{y_m} \end{pmatrix} = \boldsymbol{Q}' \begin{pmatrix} \boldsymbol{f_1} \\ \boldsymbol{f_2} \\ \vdots \\ \boldsymbol{f_m} \end{pmatrix}.$$

For pseudo QR reduction method, let the j^{th} processor perform $\boldsymbol{A_j} = \boldsymbol{X}_j'\boldsymbol{X_j}$ and $\boldsymbol{z_j} = \boldsymbol{X}_j'\boldsymbol{y_j}$ using the "online" algorithm in Figure 4.6, then the processing results from all processors can be accumulated by $\boldsymbol{A} = \sum_{j=1}^{m} \boldsymbol{A_j}$, $\boldsymbol{z} = \sum_{j=1}^{m} \boldsymbol{z_j}$ to get the results associated with the complete $\boldsymbol{X}$ for the Cholesky factorization and equation solving.

Such parallelism is called "fork-join" (see Figure 4.7 for an illustration with $m = 2$; note that $\boldsymbol{y}$

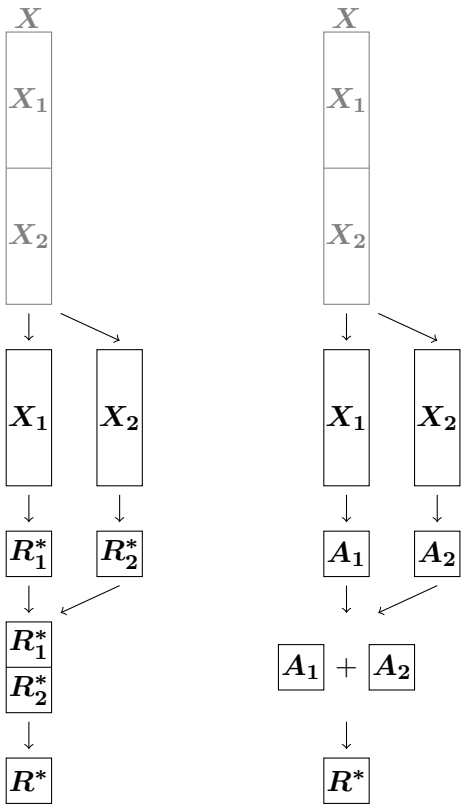


Figure 4.7: Illustration of the parallel model matrix reduction implemented in `bam`. Assuming that there are two processors, the initial model matrix $\boldsymbol{X}$ is split evenly into two row chunks $\boldsymbol{X_1}$ and $\boldsymbol{X_2}$ for processor 1 and processor 2. In QR reduction, illustrated on the left flow, a QR factorization $\boldsymbol{X_j} = \boldsymbol{Q_j R_j^*}$ is performed in parallel on two processors, then $\boldsymbol{R_1^*}$ and $\boldsymbol{R_2^*}$ are stacked by row for a last QR factorization for the final $\boldsymbol{R^*}$. In pseudo QR reduction, illustrated in the right flow, a matrix cross-product $\boldsymbol{A_j} = \boldsymbol{X_j'X_j}$ is performed on each processor, then results are accumulated $\boldsymbol{A} = \boldsymbol{A_1} + \boldsymbol{A_2}$ for a final Cholesky factorization $\boldsymbol{A} = \boldsymbol{R^{*\prime}R^*}$ to obtain $\boldsymbol{R^*}$. Computations on each processor can further use "online" algorithms in Figures 4.5 and 4.6 to keep memory footprint under control. Note that $\boldsymbol{X}$ is drawn and annotated in gray, because it is not explicitly constructed. The initial splitting and dispatching only work on row index of $\boldsymbol{X}$.

is not sketched in the Figure). It is ideal on a high-performance computing cluster, where a large model matrix can be split and dispatched to different computing nodes. The `bam` function supports such parallel processing, but since `R` software is built on a shared-memory not distributed-memory architecture, it treats each CPU core of a multi-core computer as a processor.

The chunking applied to $\boldsymbol{X}$ and $\boldsymbol{y}$ here is not to be confused with the chunking done in "online" algorithms; in practice the latter is nested in the former. Also, rows of $\boldsymbol{X}$ and $\boldsymbol{y}$ should be split roughly evenly between processors for *load balancing*. With near equal workload on each processor, all processors can finish their share at about the same time; otherwise the execution time depends on the processor that finishes its work last.

The "fork" step is where parallel computing is truely profitable; the "join" step is actually an overhead since the work is done by a single processor. Suppose that $m > 1$ processors are used at the "fork" step, we have the following observation.

- The "join" step of QR reduction factorizes an $mp \times p$ matrix. This involves $(2mp^3 - \frac{2}{3}p^3)$ FLOP which grows linearly with m.
- The "join" step of pseudo QR reduction adds up $(m-1)$ $p \times p$ matrices and performs a Cholesky factorization and equation solving. This involves $\left((m-1)p^2 + \frac{1}{3}p^3 + p^2\right)$ FLOP, which is almost constant in m for even moderately big p (since the leading order term $\frac{1}{3}p^3$ does not depend on m).

Thus one may expect that as m increases, the practical performance of parallel QR reduction would be bottlenected by the increasingly expensive "join" step.

We will get more insight if we examine *parallel speedup* or *parallel scaling factor*. Practical parallel speedup is defined with realistic execution time by

$$\text{practical parallel scaling factor} = \frac{\text{computation time using 1 core}}{\text{computation time using } m \text{ cores}}.$$

It is a measure of the effectiveness of parallel computing. Theoretical speedup is calculated with computational complexity by

$$\text{theoretical parallel scaling factor} = \frac{\text{FLOP count in serial computing}}{\text{FLOP count per core + FLOP count at "join"}}.$$

It is an upper bound for practical parallel speedup. For QR reduction this quantity is

$$\frac{2np^2 - \frac{2}{3}p^3 + 2(\frac{n}{c} - 1)p^3}{\left(2\frac{n}{m}p^2 - \frac{2}{3}p^3 + 2(\frac{n}{mc} - 1)p^3\right) + \left(2mp^3 - \frac{2}{3}p^3\right)} \approx \frac{m}{1 + \frac{m^2pc}{n(p+c)}},$$

and for pseudo QR reduction it is

$$\frac{np^2 + \frac{1}{3}p^3 + p^2}{\frac{n}{m}p^2 + \left((m-1)p^2 + \frac{1}{3}p^3 + p^2\right)} \approx \frac{m}{1 + \frac{(m-1)p}{3n+p}}.$$

These mathematical results can be summarized as follows.

- Both values are below m so perfect scaling is never possible, i.e., using m cores never reduces the execution time to $1/m$ of that in serial computing.
- The m^2 in the result for QR reduction implies that the parallel scaling for this method is likely to degrade fast as m increases. Note that having n in the denominator does not mean that the fraction would be small. The value also depends on p and c. For example, consider $n = 10^6$, $p = 5 \times 10^3$ and $c = 2 \times 10^4$, then $\frac{pc}{n(p+c)} \approx 0.0046$. When multiplied by m^2, say $8^2 = 64$, it gives nearly 0.3.
- The $\frac{(m-1)p}{3n+p}$ in the result for pseudo QR reduction also hints a performance degrade as m increases. However, it tends to have a negligible impact in practice if $n \gg p$, or even if n is just moderately bigger than p. So it is predicted that this method should have very good practical parallel scaling.

See Figure 4.8 for a practical experiment verifying those findings.

4.2.4 Summary

Below is a comparison between QR reduction and pseudo QR reduction consolidating previous demonstrations.

- QR reduction and pseudo QR reduction respectively approximately involve $2np^2$ and np^2 FLOP, thus the latter is practically 100% times faster (verified by the timing statistics for two methods in Figure 4.8 when $m = 1$);
- In "online" implementation, QR reduction has a chunking overhead that grows at a rate of $O(p^2)$, while pseudo QR reduction has no chunking overhead at all;

computation time (in seconds) for "fork-join" parallel model matrix reduction

number of cores: m		1	2	3	4	5	6	7	8
QR reduction	fork	22507	12310	8652	6205	5466	4298	3927	3269
	join	0	439	712	1053	1369	1675	1975	2268
	total	22507	12749	9363	7258	6835	5972	5902	5537
pseudo QR reduction	fork	11136	5596	3910	2817	2353	1917	1720	1463
	join	52	53	53	53	53	53	53	54
	total	11188	5649	3963	2870	2406	1970	1773	1517

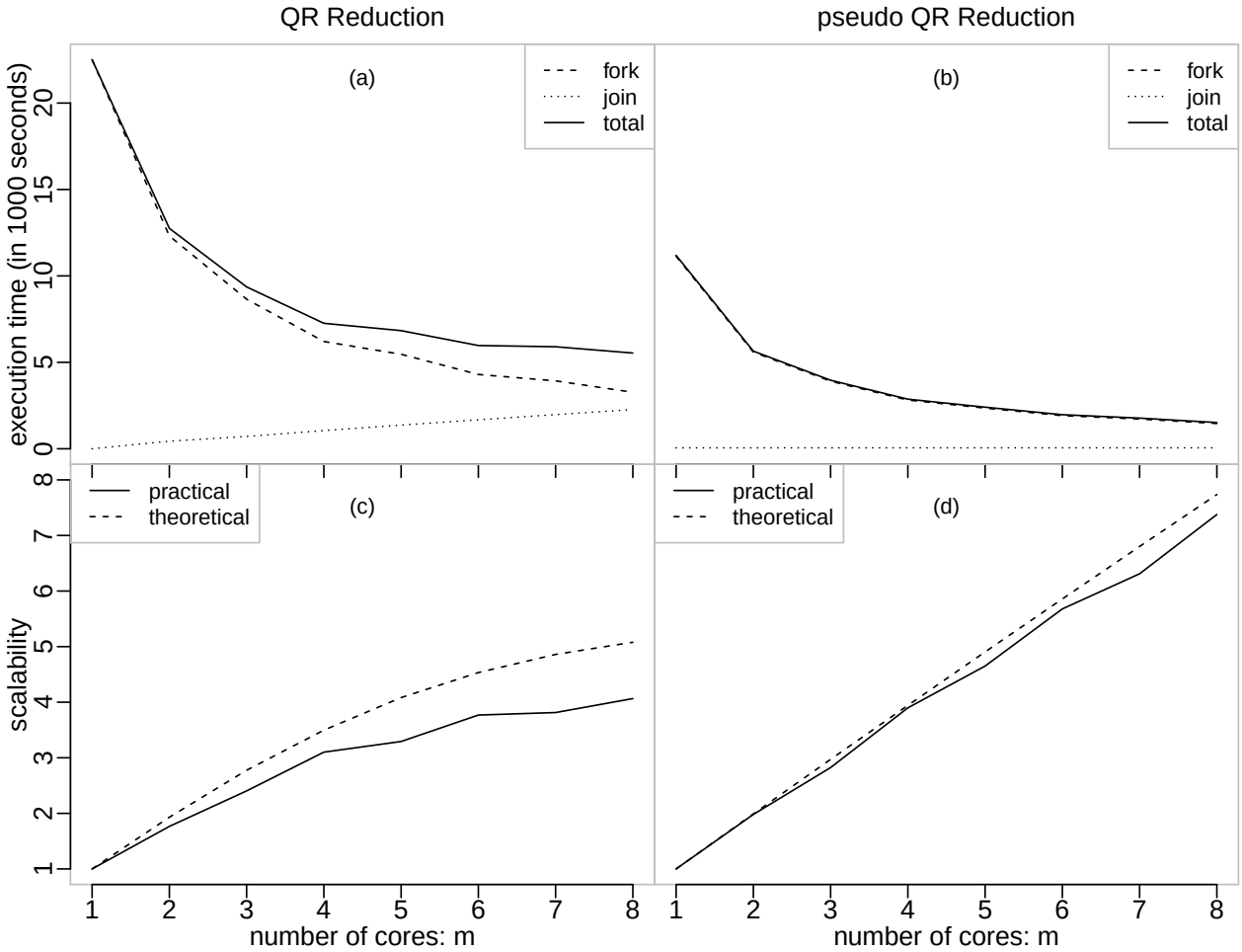


Figure 4.8: Practical performance of "fork-join" parallel QR reduction and pseudo QR reduction. Experiment is undertaken on an Intel Xeon E5-2650 v2 workstation exploiting up to 8 CPU cores (see Appendix A for hardware information). An arbitrary 430000×6300 matrix is taken as $\boldsymbol{X}$, and a chunk size $c = 10000$ is used. For example, when 8 cores are used, there are 430000 / 8 = 53750 rows in the $\boldsymbol{X_j}$ and $\boldsymbol{y_j}$ on the j^{th} core, to be processed with "online" algorithms using chunk size 10000. Panels (a) and (b) respectively sketch the computation time (total = fork + join; unit: 1000 seconds) of the two reduction methods against m (the number of cores). For parallel QR reduction, while the computation time for the "fork" step (dashed line) drops with m, the time for the "join" step (dotted line) increases linearly with m, hence the gross execution time (solid line) decreases slower than the dashed line. For parallel pseudo QR reduction, the computation time for the "join" step is low and almost constant in m, thus the solid line and the dashed line almost coincide. Panels (c) and (d) stetch parallel scaling factor of both reduction methods against m, where the dashed line is the theoretical parallel scaling factor for this example (see main text for the formula), and the solid line is the realistic parallel scaling factor. The graphs confirm that parallel QR reduction has poor scaling and parallel pseudo QR reduction has good scaling.

- In "fork-join" parallel computing, QR reduction is bottlenecked by the increasingly expensive QR factorization at "join" step as more cores are used. It gives a poor practical parallel scaling. By contrast, pseudo QR reduction has very good parallel scaling.

In short, pseudo QR reduction has performance superiority in all aspects, and should be considered as the option for estimating additive models with large datasets.

4.3 REML estimation for additive models

Completing the discussion on model matrix reduction, I will now describe computational details for the REML estimation. For additive models, $\boldsymbol{W}$ is independent of $\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}$ (hence $\boldsymbol{\lambda}$), so dropping $\log|\boldsymbol{W}|$ from $\mathcal{V}_r$ gives identical estimation of $\boldsymbol{\lambda}$ and ϕ. Therefore, let us consider the following REML score in subsequent derivation.

$$\mathcal{V}_r(\boldsymbol{\lambda},\ \phi) = (n-m)\log(2\pi\phi) + \frac{D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda})}{\phi} + \log|\boldsymbol{H}_{\boldsymbol{\lambda}}| - \log|\boldsymbol{S}_{\boldsymbol{\lambda}}|_+.$$

Let $\boldsymbol{\theta} = (\log(\boldsymbol{\lambda}),\ \log(\phi)) := (\boldsymbol{\rho},\ \nu)$, minimization of $\mathcal{V}_r$ is an unconstrained minimization w.r.t. $\boldsymbol{\theta}$. Figure 4.9 sketches a Newton-Raphson algorithm for this minimization problem, where $\boldsymbol{\theta}^{[k]}$, $\mathcal{V}_r^{[k]} = \mathcal{V}_r(\boldsymbol{\theta}^{[k]})$, $\boldsymbol{\nabla}\mathcal{V}_r^{[k]} = \frac{\partial \mathcal{V}_r}{\partial \boldsymbol{\theta}}\big|_{\boldsymbol{\theta}=\boldsymbol{\theta}^{[k]}}$ and $\boldsymbol{\nabla^2}\mathcal{V}_r^{[k]} = \frac{\partial^2 \mathcal{V}_r}{\partial\boldsymbol{\theta}\partial\boldsymbol{\theta}'}\big|_{\boldsymbol{\theta}=\boldsymbol{\theta}^{[k]}}$ are respectively the parameter vector, REML score, gradient vector and Hessian matrix at the k^{th} iteration. Computation of gradient and Hessian requires the following derivatives

$$\begin{gathered}
\frac{\partial \mathcal{V}_r}{\partial \boldsymbol{\rho}} = \frac{1}{\phi}\frac{\partial D_p}{\partial \boldsymbol{\rho}} + \frac{\partial \log|\boldsymbol{H}_{\boldsymbol{\lambda}}|}{\partial \boldsymbol{\rho}} - \frac{\partial \log|\boldsymbol{S}_{\boldsymbol{\lambda}}|_+}{\partial \boldsymbol{\rho}}, \\
\frac{\partial^2 \mathcal{V}_r}{\partial \boldsymbol{\rho}\partial \boldsymbol{\rho}'} = \frac{1}{\phi}\frac{\partial^2 D_p}{\partial \boldsymbol{\rho}\partial \boldsymbol{\rho}'} + \frac{\partial^2 \log|\boldsymbol{H}_{\boldsymbol{\lambda}}|}{\partial \boldsymbol{\rho}\partial \boldsymbol{\rho}'} - \frac{\partial^2 \log|\boldsymbol{S}_{\boldsymbol{\lambda}}|_+}{\partial \boldsymbol{\rho}\partial \boldsymbol{\rho}'}, \\
\frac{\partial \mathcal{V}_r}{\partial \nu} = n - m - \frac{D_p}{\phi}, \quad \frac{\partial^2 \mathcal{V}_r}{\partial \boldsymbol{\rho}\partial \nu} = -\frac{1}{\phi}\frac{\partial D_p}{\partial \boldsymbol{\rho}}, \quad \frac{\partial^2 \mathcal{V}_r}{\partial \nu^2} = \frac{D_p}{\phi},
\end{gathered}$$

where derivatives of $D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda})$, $\log|\boldsymbol{H}_{\boldsymbol{\lambda}}|$ and $\log|\boldsymbol{S}_{\boldsymbol{\lambda}}|_+$ w.r.t. $\boldsymbol{\rho}$ are key quantities. In particular, derivatives of $D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda})$ demand $\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}$ in the first place. In practice, we can arrange computations in each iteration into the following five steps:

1. $\boxed{\log|\boldsymbol{S}_{\boldsymbol{\lambda}}|_+}$ step computes gradient and Hessian of $\log|\boldsymbol{S}_{\boldsymbol{\lambda}}|_+$, and evaluate $\log|\boldsymbol{S}_{\boldsymbol{\lambda}}|_+$;
2. $\boxed{\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}}$ step finds $\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}} = \underset{\boldsymbol{\beta}}{\arg\min}\ D_p(\boldsymbol{\beta},\ \boldsymbol{\lambda})$ and computes $D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda})$ and $\log|\boldsymbol{H}_{\boldsymbol{\lambda}}|$;
3. $\boxed{\boldsymbol{V}_{\boldsymbol{\lambda}}}$ step computes $\boldsymbol{V}_{\boldsymbol{\lambda}} = \boldsymbol{H}_{\boldsymbol{\lambda}}^{-1}$;
4. $\boxed{D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda})}$ step computes gradient and Hessian of $D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda})$;
5. $\boxed{\log|\boldsymbol{H}_{\boldsymbol{\lambda}}|}$ step computes gradient and Hessian of $\log|\boldsymbol{H}_{\boldsymbol{\lambda}}|$.

The $\boldsymbol{V}_{\boldsymbol{\lambda}}$ above is referred to as the *unscaled covariance matrix* of $\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}$, because in Bayesian point of view (see §1.1.5), $\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}$ is an empirical Bayes estimator, whose posterior covariance matrix is $\boldsymbol{V}_{\boldsymbol{\lambda}}\phi$.

REML estimation involves substantial amount of mathematics. To avoid overwhelming you, I will demonstrate how these calculations are carried out for the simple example additive model in §1.2. For ease of reference, let me first reiterate it (by copying in (1.2) and (1.11)):

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\beta}'\boldsymbol{S}_{\boldsymbol{\lambda}}\boldsymbol{\beta} + \boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim N(\boldsymbol{0},\ \boldsymbol{W}^{-1}\phi),$$

initial value
given an initial value $\boldsymbol{\theta}^{[0]}$, compute $\mathcal{V}_r^{[0]}$, $\boldsymbol{\nabla}\mathcal{V}_r^{[0]}$, $\boldsymbol{\nabla^2}\mathcal{V}_r^{[0]}$
Newton-Raphson algorithm
$k = 1$
repeat
  Newton-type descent direction
  get a Newton step $\boldsymbol{\Delta}^{[k]} = -(\boldsymbol{\nabla^2}\mathcal{V}_r^{[k-1]})^{-1}\boldsymbol{\nabla}\mathcal{V}_r^{[k-1]}$
  $\boldsymbol{\theta}^{[k]} = \boldsymbol{\theta}^{[k-1]} + \boldsymbol{\Delta}^{[k]}$, compute $\mathcal{V}_r^{[k]}$, $\boldsymbol{\nabla}\mathcal{V}_r^{[k]}$, $\boldsymbol{\nabla^2}\mathcal{V}_r^{[k]}$
  divergence test
  while $\left(\mathcal{V}_r^{[k]} \geq \mathcal{V}_r^{[k-1]} + |\mathcal{V}_r^{[k-1]}|\epsilon\right)$
    step halving (backtracking line search)
    $\boldsymbol{\Delta}^{[k]} = \boldsymbol{\Delta}^{[k]}/2$
    $\boldsymbol{\theta}^{[k]} = \boldsymbol{\theta}^{[k-1]} + \boldsymbol{\Delta}^{[k]}$, compute $\mathcal{V}_r^{[k]}$, $\boldsymbol{\nabla}\mathcal{V}_r^{[k]}$, $\boldsymbol{\nabla^2}\mathcal{V}_r^{[k]}$
  convergence test
  if $\left(|\mathcal{V}_r^{[k]} - \mathcal{V}_r^{[k-1]}| < |\mathcal{V}_r^{[k-1]}|\epsilon\right)$ break
  $k \mathrel{+}= 1$
optimal value at convergence
$\boldsymbol{\theta}^* = \boldsymbol{\theta}^{[k]}$, $\mathcal{V}_r^* = \mathcal{V}_r^{[k]}$, $\boldsymbol{\nabla}\mathcal{V}_r^* = \boldsymbol{\nabla}\mathcal{V}_r^{[k]}$, $\boldsymbol{\nabla^2}\mathcal{V}_r^* = \boldsymbol{\nabla^2}\mathcal{V}_r^{[k]}$

Figure 4.9: Newton-Raphson algorithm for $\mathcal{V}_r$ minimization. ϵ is a small numerical tolerance for convergence and divergence test; it is a measure of relative error. A common choice is 10^{-6} or 10^{-7}.

where

$$\boldsymbol{X} = \begin{pmatrix} \boldsymbol{X_0} & \boldsymbol{X_1} & \boldsymbol{X_2} \end{pmatrix}, \quad \boldsymbol{\beta} = \begin{pmatrix} \boldsymbol{\beta_0} \\ \boldsymbol{\beta_1} \\ \boldsymbol{\beta_2} \end{pmatrix}, \quad \boldsymbol{S_\lambda} = \begin{pmatrix} \boldsymbol{0} & & \\ & \lambda_1 \boldsymbol{S_1} & \\ & & \sum_{i=1}^{q_2} \lambda_{2i} \boldsymbol{S_{2i}} \end{pmatrix}.$$

In practice having more terms in an additive model does not change the methods to be presented; one just needs a loop to apply them.

During REML estimation, we will need to compute 1st and 2nd derivatives of $\boldsymbol{S_\lambda}$ w.r.t. $\rho_1 = \log(\lambda_1)$ and $\rho_{2i} = \log(\lambda_{2i})$. This is easy, noting that smoothing parameters enter $\boldsymbol{S_\lambda}$ linearly:

$$\boldsymbol{S_\lambda} = \lambda_1 \begin{pmatrix} \boldsymbol{0} & & \\ & \boldsymbol{S_1} & \\ & & \boldsymbol{0} \end{pmatrix} + \sum_{i=1}^{q_2} \lambda_{2i} \begin{pmatrix} \boldsymbol{0} & & \\ & \boldsymbol{0} & \\ & & \boldsymbol{S_{2i}} \end{pmatrix} := \exp(\rho_1)\tilde{\boldsymbol{S}}_{\boldsymbol{1}} + \sum_{i=1}^{q_2} \exp(\rho_{2i})\tilde{\boldsymbol{S}}_{\boldsymbol{2i}}.$$

That ρ is double indexed complicates presentation of derivatives, especially the 2nd derivatives, as I have to distinguish $\partial \boldsymbol{S_\lambda}/\partial\rho_1^2$, $\partial \boldsymbol{S_\lambda}/\partial\rho_{2i}^2$, $\partial \boldsymbol{S_\lambda}/\partial\rho_{2j}^2$, $\partial \boldsymbol{S_\lambda}/\partial\rho_1\partial\rho_{2i}$, $\partial \boldsymbol{S_\lambda}/\partial\rho_1\partial\rho_{2j}$ and $\partial \boldsymbol{S_\lambda}/\partial\rho_{2i}\partial\rho_{2j}$. When necessary, I will index ρ and $\tilde{\boldsymbol{S}}$ with a single subscript and write

$$\boldsymbol{S_\lambda} := \sum_l \exp(\rho_l)\tilde{\boldsymbol{S}}_{\boldsymbol{l}}, \tag{4.3}$$

then the derivatives can be concisely presented by

$$\frac{\partial \boldsymbol{S_\lambda}}{\partial \rho_i} = \exp(\rho_i)\tilde{\boldsymbol{S}}_{\boldsymbol{i}}, \quad \frac{\partial^2 \boldsymbol{S_\lambda}}{\partial \rho_i \partial \rho_j} = \delta_i^j \exp(\rho_i)\tilde{\boldsymbol{S}}_{\boldsymbol{i}} = \delta_i^j \frac{\partial \boldsymbol{S_\lambda}}{\partial \rho_i}. \tag{4.4}$$

In the three derivative-related computation steps, double indexing style for smoothing parameters is convenient for $\boxed{\log|\boldsymbol{S_\lambda}|_+}$ step, whereas single indexing style is convenient for $\boxed{D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda})}$ and $\boxed{\log|\boldsymbol{H_\lambda}|}$ steps. Readers should bear this in mind as in the forthcoming mathematical derivations for those computation steps, such indexing convention is not to be reiterated.

The rest of this section is arranged as such.

- §4.3.1 explains an initial reparametrization to $\boldsymbol{\beta}$, which helps reduce computational complexity of the five computation steps.

- §4.3.2 to §4.3.6 cover computational details for the five computation steps.
- §4.3.7 elaborates a special numerical issue: the stable evaluation of $\log|\boldsymbol{S}_{\boldsymbol{\lambda}}|_+$.
- §4.3.8 summarizes FLOP count for the five computation steps and mentions the necessary post-processing to reverse the initial reparametrization when computing $\hat{\boldsymbol{y}}$ or making out-of-sample prediction.

4.3.1 Initial reparametrization

This section demonstrates the initial parametrization to the example additive model (1.10) at the start of its REML estimation. Let p_i be the number of columns of $\boldsymbol{X_i}$ (so that $\sum_i p_i = p$) and m_i be the null space dimension of $\boldsymbol{S_i}$ (or the number of $\boldsymbol{S_i}$'s zero eigenvalues). I will show how to reparametrize $\boldsymbol{\beta_1}$ and $\boldsymbol{\beta_2}$, as well as how $\boldsymbol{S_\lambda}$ and $\boldsymbol{R^*}$ (see (4.2)) are transformed accordingly.

Consider $\boldsymbol{f_1}$ first. After an eigen decomposition $\boldsymbol{S_1} = \boldsymbol{U_1 D_1 U_1'}$, the trailing m_1 eigenvalues on $\boldsymbol{D_1}$'s main diagonal are zeros. Let $\dot{\boldsymbol{D}}_{\boldsymbol{1}}$ be a modified $\boldsymbol{D_1}$ where these zero eigenvalues are replaced by ones, and $\dot{\boldsymbol{I}}_{\boldsymbol{1}}$ be a modified identity matrix where the trailing m_1 ones on its main diagonal are set to zeros, then the eigen decomposition can be written as $\boldsymbol{S_1} = (\boldsymbol{U_1}\dot{\boldsymbol{D}}_{\boldsymbol{1}}^{\frac{1}{2}})\dot{\boldsymbol{I}}_{\boldsymbol{1}}(\boldsymbol{U_1}\dot{\boldsymbol{D}}_{\boldsymbol{1}}^{\frac{1}{2}})'$. Defining a reparametrization $\dot{\boldsymbol{\beta}}_{\boldsymbol{1}} = \dot{\boldsymbol{D}}_{\boldsymbol{1}}^{\frac{1}{2}}\boldsymbol{U_1'}\boldsymbol{\beta_1}$, the quadratic penalty $\lambda_1\boldsymbol{\beta_1'S_1\beta_1}$ reduces to $\lambda_1\dot{\boldsymbol{\beta}}_{\boldsymbol{1}}'\dot{\boldsymbol{I}}_{\boldsymbol{1}}\dot{\boldsymbol{\beta}}_{\boldsymbol{1}}$ that resembles a ridge penalty. In fact, this reparametrization makes the link between smoothing models and mixed models more evident, as the first $(p_1 - m_1)$ and the last m_1 elements of $\dot{\boldsymbol{\beta}}_{\boldsymbol{1}}$ can be respectively interpreted as an i.i.d. random effect (which is penalized) and some fixed effect (which is not penalized).

For $\boldsymbol{f_2}$, consider an eigen decomposition $\sum_{i=1}^{q_2}\boldsymbol{S_{2i}} = \boldsymbol{U_2 D_2 U_2'}$. As $\boldsymbol{S_2} = \sum_{i=1}^{q_2}\lambda_{2i}\boldsymbol{S_{2i}}$ has null space dimension m_2 (for any λ_{2i}), the last m_2 eigenvalues in $\boldsymbol{D_2}$ are zeros. Define a reparametrization $\dot{\boldsymbol{\beta}}_{\boldsymbol{2}} = \boldsymbol{U_2'}\boldsymbol{\beta_2}$, then the first $(p_2 - m_2)$ and the last m_2 elements of $\dot{\boldsymbol{\beta}}_{\boldsymbol{2}}$ can be respectively regarded as some random effect (which is penalized) and some fixed effect (which is not penalized). Let $\bar{\boldsymbol{U}}_{\boldsymbol{2}} = \boldsymbol{U_2}(,\ 1:(p_2 - m_2))$, then $\dot{\boldsymbol{S}}_{\boldsymbol{2}} = \sum_{i=1}^{q_2}\lambda_{2i}\bar{\boldsymbol{S}}_{\boldsymbol{2i}}$ is a reasonable full-rank penalty for the random effect, where $\bar{\boldsymbol{S}}_{\boldsymbol{2i}} = \bar{\boldsymbol{U}}_{\boldsymbol{2}}'\boldsymbol{S_{2i}}\bar{\boldsymbol{U}}_{\boldsymbol{2}}$ is a projection of $\boldsymbol{S_{2i}}$ onto $\boldsymbol{S_2}$'s column space.

In above demonstration it is assumed that the null space dimension m_i of $\boldsymbol{S_i}$ is known a priori. While this is true for most spline basis, it can also be numerically determined otherwise, because eigen decomposition is rank revealing. Either way, the reparametrizations above for $\boldsymbol{f_1}$ and $\boldsymbol{f_2}$ can be summarized by a single transformation $\dot{\boldsymbol{\beta}} = \boldsymbol{U\beta}$, where

$$\boldsymbol{U} = \begin{pmatrix} \boldsymbol{I} & & \\ & \dot{\boldsymbol{D}}_{\boldsymbol{1}}^{\frac{1}{2}}\boldsymbol{U_1'} & \\ & & \boldsymbol{U_2'} \end{pmatrix}, \quad \boldsymbol{U}^{-1} = \begin{pmatrix} \boldsymbol{I} & & \\ & \boldsymbol{U_1}\dot{\boldsymbol{D}}_{\boldsymbol{1}}^{-\frac{1}{2}} & \\ & & \boldsymbol{U_2} \end{pmatrix}.$$

With the inverse transformation $\boldsymbol{\beta} = \boldsymbol{U}^{-1}\dot{\boldsymbol{\beta}}$, the penalized weighted residual sum of squares $D_p(\boldsymbol{\beta},\ \boldsymbol{\lambda})$ (see (4.2) if you forgot what it is) can be rewritten as

$$D_p(\dot{\boldsymbol{\beta}},\ \boldsymbol{\lambda}) = \left\|\boldsymbol{f} - \dot{\boldsymbol{R}}^*\dot{\boldsymbol{\beta}}\right\|^2 + \dot{\boldsymbol{\beta}}'\dot{\boldsymbol{S}}_{\boldsymbol{\lambda}}\dot{\boldsymbol{\beta}} + \left\|\boldsymbol{W}^{\frac{1}{2}}\boldsymbol{y}\right\|^2 - \|\boldsymbol{f}\|^2,$$

where $\dot{\boldsymbol{R}}^* = \boldsymbol{R}^*\boldsymbol{U}^{-1}$ and

$$\dot{\boldsymbol{S}}_{\boldsymbol{\lambda}} = \begin{pmatrix} \boldsymbol{0} & & \\ & \begin{bmatrix} \lambda_1\boldsymbol{I} & \\ & \boldsymbol{0} \end{bmatrix} & \\ & & \begin{bmatrix} \sum_{i=1}^{q_2}\lambda_{2i}\bar{\boldsymbol{S}}_{\boldsymbol{2i}} & \\ & \boldsymbol{0} \end{bmatrix} \end{pmatrix}. \tag{4.5}$$

The initial reparametrization is most intuitive in terms of the similarity transformation on $\boldsymbol{S_\lambda}$. For convenience, depending on how the penalty matrix $\boldsymbol{S_i}$ of a spline $\boldsymbol{f_i}$ enters $\boldsymbol{S_\lambda}$, let us call $\boldsymbol{S_i}$ a *single-λ*

block, if it has only one smoothing parameter (like $\boldsymbol{S_1}$ in the example), and a *multi-λ block* otherwise (like $\boldsymbol{S_2}$ in the example). Then in general, the initial reparametrization does the following.

- A single-λ block is transformed to an identity-alike block, whose main diagonal elements (before multiplying λ) are either one or zero. A special case is when this block is readily diagonal (for example, the ridge penalty of an i.i.d. random effect), then nothing but a scaling is needed.
- A multi-λ block (usually rank-deficient) is projected onto a full-rank multi-λ block with a smaller dimension, and a trailing block of zeros.

The initial reparametrization is useful in two ways.

Firstly, computating $|\dot{\boldsymbol{S}}_{\boldsymbol{\lambda}}|_+$ is more straightforward than computing $|\boldsymbol{S}_{\boldsymbol{\lambda}}|_+$, because all non-zero diagonal blocks in $\dot{\boldsymbol{S}}_{\boldsymbol{\lambda}}$ have full rank, while similar blocks in $\boldsymbol{S}_{\boldsymbol{\lambda}}$ do not. As a result, we have

$$|\dot{\boldsymbol{S}}_{\boldsymbol{\lambda}}|_+ = \lambda_1^{(p_1-m_1)}|\bar{\boldsymbol{S}}_{\boldsymbol{2}}|, \quad \log|\dot{\boldsymbol{S}}_{\boldsymbol{\lambda}}|_+ = (p_1-m_1)\rho_1 + \log|\bar{\boldsymbol{S}}_{\boldsymbol{2}}|. \tag{4.6}$$

However, it is worth emphasizing that $|\dot{\boldsymbol{S}}_{\boldsymbol{\lambda}}|_+ \neq |\boldsymbol{S}_{\boldsymbol{\lambda}}|_+$, as they are respectively the log-determinant in the REML score parametrized with $\dot{\boldsymbol{\beta}}$ and $\boldsymbol{\beta}$. To be precise, after the initial reparametrization, we estimate $\dot{\boldsymbol{\beta}}$ not $\boldsymbol{\beta}$ in Newton-Raphson iterations. We back transform $\dot{\boldsymbol{\beta}}$ to $\boldsymbol{\beta}$ after convergence of the iterations. I will mention this issue again in §4.3.8.

Secondly, it makes computation more efficient. $\boxed{\log|\boldsymbol{H}_{\boldsymbol{\lambda}}|}$ step will need to compute $\tilde{\boldsymbol{S}}_{\boldsymbol{i}}\boldsymbol{V}_{\boldsymbol{\lambda}}$ (see (4.3) if you forgot what $\tilde{\boldsymbol{S}}_{\boldsymbol{i}}$ is). Without the initial parametrization, $\tilde{\boldsymbol{S}}_{\boldsymbol{i}}$ is either of the following (double indexing style is used to distinguish a single-λ block and a multi-λ block):

$$\tilde{\boldsymbol{S}}_{\boldsymbol{1}} = \begin{pmatrix} \boldsymbol{0} & & \\ & \boldsymbol{S_1} & \\ & & \boldsymbol{0} \end{pmatrix}, \quad \tilde{\boldsymbol{S}}_{\boldsymbol{2i}} = \begin{pmatrix} \boldsymbol{0} & & \\ & \boldsymbol{0} & \\ & & \boldsymbol{S_{2i}} \end{pmatrix},$$

whereas after the initial reparametrization it is either of the following:

$$\tilde{\boldsymbol{S}}_{\boldsymbol{1}} = \begin{pmatrix} \boldsymbol{0} & & \\ & \begin{bmatrix} \boldsymbol{I} & \boldsymbol{0} \end{bmatrix} & \\ & & \boldsymbol{0} \end{pmatrix}, \quad \tilde{\boldsymbol{S}}_{\boldsymbol{2i}} = \begin{pmatrix} \boldsymbol{0} & & \\ & \boldsymbol{0} & \\ & & \begin{bmatrix} \bar{\boldsymbol{S}}_{\boldsymbol{2i}} & \boldsymbol{0} \end{bmatrix} \end{pmatrix}.$$

The matrix multiplication $\tilde{\boldsymbol{S}}_{\boldsymbol{1}}\boldsymbol{V}_{\boldsymbol{\lambda}}$ is just a row extraction on $\boldsymbol{V}_{\boldsymbol{\lambda}}$ in the latter case, but involves a dense matrix-matrix multiplication in the former.

To simplify notation, I will hereafter drop the '$\cdot$' from $\dot{\boldsymbol{\beta}}$, $\dot{\boldsymbol{R}}^*$ and $\dot{\boldsymbol{S}}_{\boldsymbol{\lambda}}$ from §4.3.2 to §4.3.7, assuming that $\boldsymbol{\beta}$, $\boldsymbol{R}^*$ and $\boldsymbol{S}_{\boldsymbol{\lambda}}$ have already absorbed the initial reparametrization.

4.3.2 Derivatives of $\log|\boldsymbol{S}_{\boldsymbol{\lambda}}|_+$

Following (4.6), derivatives w.r.t. ρ_1 are

$$\frac{\partial \log|\boldsymbol{S}_{\boldsymbol{\lambda}}|_+}{\partial \rho_1} = (p_1 - m_1), \qquad \frac{\partial^2 \log|\boldsymbol{S}_{\boldsymbol{\lambda}}|_+}{\partial \rho_1^2} = 0.$$

For derivatives w.r.t. $\boldsymbol{\rho_2} = (\rho_{21}, \cdots, \rho_{2q_2})$, define $\boldsymbol{D_{2i}} = \bar{\boldsymbol{S}}_{\boldsymbol{2}}^{-1}\frac{\partial \bar{\boldsymbol{S}}_{\boldsymbol{2}}}{\partial \rho_{2i}} = \exp(\rho_{2i})\bar{\boldsymbol{S}}_{\boldsymbol{2}}^{-1}\bar{\boldsymbol{S}}_{\boldsymbol{2i}}$, then using Harville (1997, equation (8.6), §15.8; equation (9.3), §15.9), there are

$$\frac{\partial \log|\boldsymbol{S}_{\boldsymbol{\lambda}}|_+}{\partial \rho_{2i}} = \frac{\partial \log|\bar{\boldsymbol{S}}_{\boldsymbol{2}}|}{\partial \rho_{2i}} = \mathtt{tr}(\boldsymbol{D_{2i}}),$$

$$\frac{\partial^2 \log|\boldsymbol{S}_{\boldsymbol{\lambda}}|_+}{\partial \rho_{2i}\partial \rho_{2j}} = \frac{\partial^2 \log|\bar{\boldsymbol{S}}_{\boldsymbol{2}}|}{\partial \rho_{2i}\partial \rho_{2j}} = \delta_i^j \mathtt{tr}(\boldsymbol{D_{2i}}) - \mathtt{tr}(\boldsymbol{D_{2i}}\boldsymbol{D_{2j}}),$$

which can be very efficiently computed using

$$\mathtt{tr}(\boldsymbol{D_{2i}}) = \sum_{s=1}^{p} \boldsymbol{D_{2i}}(s,\ s), \quad \mathtt{tr}(\boldsymbol{D_{2i}}\boldsymbol{D_{2j}}) = \sum_{s=1}^{p}\sum_{t=1}^{p} \boldsymbol{D_{2i}}(s,\ t)\boldsymbol{D'_{2j}}(s,\ t).$$

Furthermore, there are no 2nd derivatives "between blocks", i.e.,

$$\frac{\partial^2 \log|\boldsymbol{S_\lambda}|_+}{\partial \boldsymbol{\rho_2}\partial\rho_1} = \boldsymbol{0}.$$

As a result, the Hessian matrix for $\log|\boldsymbol{S_\lambda}|_+$ is

$$\frac{\partial^2 \log|\boldsymbol{S_\lambda}|_+}{\partial \boldsymbol{\rho}\partial\boldsymbol{\rho'}} = \begin{pmatrix} 0 & \\ & \frac{\partial^2 \log|\bar{\boldsymbol{S}}_{\boldsymbol{2}}|}{\partial \boldsymbol{\rho_2}\partial\boldsymbol{\rho'_2}} \end{pmatrix}.$$

Computation of the key quantity $\boldsymbol{D_{2i}}$ does not require an explicit matrix inverse $\bar{\boldsymbol{S}}_{\boldsymbol{2}}^{-1}$. Either a QR factorization or a Cholesky factorization can be used.

- A pivoted QR factorization (see §4.1.1 if you need a revision) $\bar{\boldsymbol{S}}_{\boldsymbol{2}} = \boldsymbol{Q_2}\boldsymbol{R_2^*}$ yields $\boldsymbol{D_{2i}} = \boldsymbol{R_2^{*-1}}(\boldsymbol{Q'_2}\bar{\boldsymbol{S}}_{\boldsymbol{2i}})$. Computation of the orthonormal transform can be done by a sequence of Householder transformations, as is explained in §4.1.1. This involves $2p_2^3$ FLOP. Solving the subsequent triangular system of linear equations (see §4.1.3 if you need a revision) involves p_2^3 FLOP. Since there are q_2 smoothing parameters in $\bar{\boldsymbol{S}}_{\boldsymbol{2}}$, the overall FLOP count for computing gradient and Hessian of this multi-λ block is $(3q_2p_2^3 + \frac{4}{3}p_2^3)$, where the additional $\frac{4}{3}p_2^3$ FLOP come from the initial QR factorization.
- Since $\bar{\boldsymbol{S}}_{\boldsymbol{2}}$ is a positive-definite matrix, a more efficient method is to compute its pivoted Cholesky factorization (with pre-conditioning) (see §4.1.2 if you need a revision) $\bar{\boldsymbol{S}}_{\boldsymbol{2}} = \boldsymbol{R_2^{*\prime}}\boldsymbol{R_2^*}$, then obtain $\boldsymbol{D_{2i}}$ by solving two triangular systems $\boldsymbol{D_{2i}} = \boldsymbol{R_2^{*-1}}(\boldsymbol{R_2^{*\prime-1}}\bar{\boldsymbol{S}}_{\boldsymbol{2i}})$. This would reduce the overall complexity to $(2q_2p_2^3 + \frac{1}{3}p_2^3)$ FLOP.

Given $\boldsymbol{R_2^*}$, the log-determinant $\log|\bar{\boldsymbol{S}}_{\boldsymbol{2}}|$ can be easily evaluated.

- For the QR apporach with $\boldsymbol{R_2^*} = \boldsymbol{R_2}\boldsymbol{P'_2}$, there is $\log|\bar{\boldsymbol{S}}_{\boldsymbol{2}}| = \sum_{i=1}^{p_2}\log|\boldsymbol{R_2}(i,\ i)|$.
- For the Cholesky approach with $\boldsymbol{R_2^*} = \boldsymbol{R_2}\boldsymbol{P'_2}\boldsymbol{J_2}$, there is $\log|\bar{\boldsymbol{S}}_{\boldsymbol{2}}| = 2\sum_{i=1}^{p_2}\big(\log \boldsymbol{R_2}(i,\ i) + \log \boldsymbol{J_2}(i,\ i)\big)$.

However, the result does not always end up correct. This is a special numerical issue with a different flavour from other aspects of REML estimation. I will skip it for now and explain its stable evaluation in §4.3.7.

4.3.3 Estimation of $\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}$

Consider a pivoted Cholesky factorization (see §4.1.2 if you need a revision) for the penalty matrix $\boldsymbol{S_\lambda} = \boldsymbol{E_\lambda^{*\prime}}\boldsymbol{E_\lambda^*}$. Note that since $\boldsymbol{S_\lambda}$ is block diagonal (see (4.5) if you forgot what it is), there is

$$\boldsymbol{E_\lambda^*} = \begin{pmatrix} \boldsymbol{0} & & \\ & \begin{bmatrix} \lambda_1^{\frac{1}{2}}\boldsymbol{I} & \\ & \boldsymbol{0} \end{bmatrix} & \\ & & \begin{bmatrix} \boldsymbol{E_2^*} & \\ & \boldsymbol{0} \end{bmatrix} \end{pmatrix}, \quad \bar{\boldsymbol{S}}_{\boldsymbol{2}} = \boldsymbol{E_2^{*\prime}}\boldsymbol{E_2^*}.$$

In fact, $\boldsymbol{E}_{\boldsymbol{\lambda}}^*$ will be readily available, if Cholesky method has been used for computing derivatives in the $\boxed{\log|\boldsymbol{S}_{\boldsymbol{\lambda}}|_+}$ step. For example, $\boldsymbol{E}_{\boldsymbol{2}}^*$ here is just $\boldsymbol{R}_{\boldsymbol{2}}^*$. Clearly there are $(p-p_0-m_1-m_2)$ number of rows of zeros, which can be dropped from $\boldsymbol{E}_{\boldsymbol{\lambda}}^*$. Now, $D_p(\boldsymbol{\beta},\ \boldsymbol{\lambda})$ (see (4.2) if you forgot what it is) can be rewritten as

$$\left\| \begin{pmatrix} \boldsymbol{f} \\ \boldsymbol{0} \end{pmatrix} - \begin{pmatrix} \boldsymbol{R}^* \\ \boldsymbol{E}_{\boldsymbol{\lambda}}^* \end{pmatrix} \boldsymbol{\beta} \right\|^2 + \left\| \boldsymbol{W}^{\frac{1}{2}} \boldsymbol{y} \right\|^2 - \|\boldsymbol{f}\|^2.$$

To find its minimizer, take a thin pivoted QR factorization (see §4.1.1 if you need a revision)

$$\begin{pmatrix} \boldsymbol{R}^* \\ \boldsymbol{E}_{\boldsymbol{\lambda}}^* \end{pmatrix} = \boldsymbol{Q}_{\boldsymbol{\lambda}} \boldsymbol{R}_{\boldsymbol{\lambda}}^* = \boldsymbol{Q}_{\boldsymbol{\lambda}} \boldsymbol{R}_{\boldsymbol{\lambda}} \boldsymbol{P}_{\boldsymbol{\lambda}}',$$

then the objective function becomes

$$\left\| \boldsymbol{f}_{\boldsymbol{\lambda}} - \boldsymbol{R}_{\boldsymbol{\lambda}}^* \boldsymbol{\beta} \right\|^2 + \left\| \boldsymbol{W}^{\frac{1}{2}} \boldsymbol{y} \right\|^2 - \|\boldsymbol{f}_{\boldsymbol{\lambda}}\|^2, \quad \boldsymbol{f}_{\boldsymbol{\lambda}} = \boldsymbol{Q}_{\boldsymbol{\lambda}}' \begin{pmatrix} \boldsymbol{f} \\ \boldsymbol{0} \end{pmatrix}.$$

Its minimizer of is $\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}} = \boldsymbol{R}_{\boldsymbol{\lambda}}^{*-1} \boldsymbol{f}_{\boldsymbol{\lambda}}$ which can be computed by solving a triangular system of linear equations (see §4.1.3 if you need a revision). Meanwhile, $D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda}) = \left\| \boldsymbol{W}^{\frac{1}{2}} \boldsymbol{y} \right\|^2 - \|\boldsymbol{f}_{\boldsymbol{\lambda}}\|^2$. Note that the QR factorization implies a pivoted Cholesky factorization $\boldsymbol{H}_{\boldsymbol{\lambda}} = \boldsymbol{R}^{*\prime} \boldsymbol{R}^* + \boldsymbol{S}_{\boldsymbol{\lambda}} = (\boldsymbol{Q}_{\boldsymbol{\lambda}} \boldsymbol{R}_{\boldsymbol{\lambda}}^*)' \boldsymbol{Q}_{\boldsymbol{\lambda}} \boldsymbol{R}_{\boldsymbol{\lambda}}^* = \boldsymbol{R}_{\boldsymbol{\lambda}}^{*\prime} \boldsymbol{R}_{\boldsymbol{\lambda}}^*$, hence, $\log|\boldsymbol{H}_{\boldsymbol{\lambda}}| = 2\sum_{i=1}^{p} \log|\boldsymbol{R}_{\boldsymbol{\lambda}}(i,\ i)|$.

Sometimes $\boldsymbol{R}_{\boldsymbol{\lambda}}$ can be singular with rank $r < p$. In this case, initialize $\boldsymbol{z}$ as a length-p vector of zeros and set $\boldsymbol{z}(1:r) = \boldsymbol{R}_{\boldsymbol{\lambda}}(1:r,\ 1:r)^{-1} \boldsymbol{f}_{\boldsymbol{\lambda}}(1:r)$, then compute $\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}} = \boldsymbol{P}_{\boldsymbol{\lambda}} \boldsymbol{z}$, $D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda}) = \left\| \boldsymbol{W}^{\frac{1}{2}} \boldsymbol{y} \right\|^2 - \|\boldsymbol{f}_{\boldsymbol{\lambda}}(1:r)\|^2$ and $\log|\boldsymbol{H}_{\boldsymbol{\lambda}}| = \sum_{i=1}^{r} \log|\boldsymbol{R}_{\boldsymbol{\lambda}}(i,\ i)|$.

Computations in this step are dominated by the QR factorization of the (approximately) $2p \times p$ augmented model matrix, which involves $\frac{10}{3}p^3$ FLOP.

4.3.4 Computing unscaled covariance matrix $\boldsymbol{V}_{\boldsymbol{\lambda}}$

Given the Cholesky factorization $\boldsymbol{H}_{\boldsymbol{\lambda}} = \boldsymbol{R}_{\boldsymbol{\lambda}}^{*\prime} \boldsymbol{R}_{\boldsymbol{\lambda}}^*$, its inverse $\boldsymbol{V}_{\boldsymbol{\lambda}}$ can be computed using methods described in §4.1.4 with $\frac{2}{3}p^3$ FLOP. In case $\boldsymbol{R}_{\boldsymbol{\lambda}}^*$ is singular with rank $r < p$, initialize $\boldsymbol{Z}$ as a $p \times p$ matrix of zeros and set $\boldsymbol{Z}(1:r,\ 1:r) = \left(\boldsymbol{R}_{\boldsymbol{\lambda}}(1:r,\ 1:r)' \boldsymbol{R}_{\boldsymbol{\lambda}}(1:r,\ 1:r)\right)^{-1}$, then compute $\boldsymbol{V}_{\boldsymbol{\lambda}} = \boldsymbol{P}_{\boldsymbol{\lambda}} \boldsymbol{Z} \boldsymbol{P}_{\boldsymbol{\lambda}}'$ by row and column permutations.

4.3.5 Derivatives of $D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda})$

As $D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda}) = D(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}) + \hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}' \boldsymbol{S}_{\boldsymbol{\lambda}} \hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}$, we consider derivatives of the two parts separately.

Using chain rule and product rule, the 1st and 2nd derivatives of $D(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}})$ w.r.t. $\boldsymbol{\rho}$ are

$$\frac{\partial D}{\partial \rho_i} = \frac{\partial \hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}'}{\partial \rho_i} \frac{\partial D}{\partial \hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}}, \quad \frac{\partial^2 D}{\partial \rho_i \partial \rho_j} = \frac{\partial^2 \hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}'}{\partial \rho_i \partial \rho_j} \frac{\partial D}{\partial \hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}} + \frac{\partial \hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}'}{\partial \rho_i} \frac{\partial^2 D}{\partial \hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}} \partial \hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}'} \frac{\partial \hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}}{\partial \rho_j}.$$

Among these quantities, derivatives of $D(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}})$ w.r.t. $\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}$ can be obtained from $D(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}) = D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda}) - \hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}' \boldsymbol{S}_{\boldsymbol{\lambda}} \hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}$. Knowing (since $\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}$ is the minimizer of D_p; see (1.6) if you need a revision)

$$\frac{\partial D_p}{\partial \hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}} = \boldsymbol{0}, \quad \frac{\partial^2 D_p}{\partial \hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}} \partial \hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}'} = 2\boldsymbol{H}_{\boldsymbol{\lambda}},$$

as well as

$$\frac{\partial \hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}' \boldsymbol{S}_{\boldsymbol{\lambda}} \hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}}{\partial \hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}} = 2\boldsymbol{S}_{\boldsymbol{\lambda}} \hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}, \quad \frac{\partial \hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}' \boldsymbol{S}_{\boldsymbol{\lambda}} \hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}}{\partial \hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}} \partial \hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}'} = 2\boldsymbol{S}_{\boldsymbol{\lambda}},$$

we have

$$\frac{\partial D}{\partial \hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}} = -2\boldsymbol{S}_{\boldsymbol{\lambda}}\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}, \qquad \frac{\partial^2 D}{\partial \hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}} \partial \hat{\boldsymbol{\beta}}'_{\boldsymbol{\lambda}}} = 2(\boldsymbol{H}_{\boldsymbol{\lambda}} - \boldsymbol{S}_{\boldsymbol{\lambda}}).$$

Thus,

$$\frac{\partial D}{\partial \rho_i} = -2\frac{\partial \hat{\boldsymbol{\beta}}'_{\boldsymbol{\lambda}}}{\partial \rho_i}\boldsymbol{S}_{\boldsymbol{\lambda}}\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}, \qquad \frac{\partial^2 D}{\partial \rho_i \partial \rho_j} = -2\frac{\partial^2 \hat{\boldsymbol{\beta}}'_{\boldsymbol{\lambda}}}{\partial \rho_i \partial \rho_j}\boldsymbol{S}_{\boldsymbol{\lambda}}\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}} + 2\frac{\partial \hat{\boldsymbol{\beta}}'_{\boldsymbol{\lambda}}}{\partial \rho_i}(\boldsymbol{H}_{\boldsymbol{\lambda}} - \boldsymbol{S}_{\boldsymbol{\lambda}})\frac{\partial \hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}}{\partial \rho_j}.$$

Now, using product rule, the 1st and 2nd derivatives of $\hat{\boldsymbol{\beta}}'_{\boldsymbol{\lambda}}\boldsymbol{S}_{\boldsymbol{\lambda}}\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}$ are

$$\begin{aligned}
\frac{\partial \hat{\boldsymbol{\beta}}'_{\boldsymbol{\lambda}}\boldsymbol{S}_{\boldsymbol{\lambda}}\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}}{\partial \rho_i} &= \frac{\partial \hat{\boldsymbol{\beta}}'_{\boldsymbol{\lambda}}}{\partial \rho_i}\boldsymbol{S}_{\boldsymbol{\lambda}}\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}} + \hat{\boldsymbol{\beta}}'_{\boldsymbol{\lambda}}\boldsymbol{S}_{\boldsymbol{\lambda}}\frac{\partial \hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}}{\partial \rho_i} + \hat{\boldsymbol{\beta}}'_{\boldsymbol{\lambda}}\frac{\partial \boldsymbol{S}_{\boldsymbol{\lambda}}}{\partial \rho_i}\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}} = 2\frac{\partial \hat{\boldsymbol{\beta}}'_{\boldsymbol{\lambda}}}{\partial \rho_i}\boldsymbol{S}_{\boldsymbol{\lambda}}\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}} + \hat{\boldsymbol{\beta}}'_{\boldsymbol{\lambda}}\frac{\partial \boldsymbol{S}_{\boldsymbol{\lambda}}}{\partial \rho_i}\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}, \\
\frac{\partial^2 \hat{\boldsymbol{\beta}}'_{\boldsymbol{\lambda}}\boldsymbol{S}_{\boldsymbol{\lambda}}\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}}{\partial \rho_i \partial \rho_j} &= 2\Big(\frac{\partial^2 \hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}}{\partial \rho_i \partial \rho_j}\boldsymbol{S}_{\boldsymbol{\lambda}}\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}} + \frac{\partial \hat{\boldsymbol{\beta}}'_{\boldsymbol{\lambda}}}{\partial \rho_i}\frac{\partial \boldsymbol{S}_{\boldsymbol{\lambda}}}{\partial \rho_j}\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}} + \frac{\partial \hat{\boldsymbol{\beta}}'_{\boldsymbol{\lambda}}}{\partial \rho_i}\boldsymbol{S}_{\boldsymbol{\lambda}}\frac{\partial \hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}}{\partial \rho_j}\Big) + \\
&\quad \frac{\partial \hat{\boldsymbol{\beta}}'_{\boldsymbol{\lambda}}}{\partial \rho_j}\frac{\partial \boldsymbol{S}_{\boldsymbol{\lambda}}}{\partial \rho_i}\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}} + \hat{\boldsymbol{\beta}}'_{\boldsymbol{\lambda}}\frac{\partial^2 \boldsymbol{S}_{\boldsymbol{\lambda}}}{\partial \rho_i \partial \rho_j}\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}} + \hat{\boldsymbol{\beta}}'_{\boldsymbol{\lambda}}\frac{\partial \boldsymbol{S}_{\boldsymbol{\lambda}}}{\partial \rho_i}\frac{\partial \hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}}{\partial \rho_j} \\
&= 2\Big(\frac{\partial^2 \hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}}{\partial \rho_i \partial \rho_j}\boldsymbol{S}_{\boldsymbol{\lambda}}\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}} + \frac{\partial \hat{\boldsymbol{\beta}}'_{\boldsymbol{\lambda}}}{\partial \rho_i}\frac{\partial \boldsymbol{S}_{\boldsymbol{\lambda}}}{\partial \rho_j}\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}} + \frac{\partial \hat{\boldsymbol{\beta}}'_{\boldsymbol{\lambda}}}{\partial \rho_i}\boldsymbol{S}_{\boldsymbol{\lambda}}\frac{\partial \hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}}{\partial \rho_j} + \frac{\partial \hat{\boldsymbol{\beta}}'_{\boldsymbol{\lambda}}}{\partial \rho_j}\frac{\partial \boldsymbol{S}_{\boldsymbol{\lambda}}}{\partial \rho_i}\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}\Big) + \\
&\quad \hat{\boldsymbol{\beta}}'_{\boldsymbol{\lambda}}\frac{\partial^2 \boldsymbol{S}_{\boldsymbol{\lambda}}}{\partial \rho_i \partial \rho_j}\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}.
\end{aligned}$$

When combining results from the two parts, many terms cancel out and we have

$$\begin{aligned}
\frac{\partial D_p}{\partial \rho_i} &= \hat{\boldsymbol{\beta}}'_{\boldsymbol{\lambda}}\frac{\partial \boldsymbol{S}_{\boldsymbol{\lambda}}}{\partial \rho_i}\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}, \\
\frac{\partial^2 D_p}{\partial \rho_i \partial \rho_j} &= 2\Big(\frac{\partial \hat{\boldsymbol{\beta}}'_{\boldsymbol{\lambda}}}{\partial \rho_i}\frac{\partial \boldsymbol{S}_{\boldsymbol{\lambda}}}{\partial \rho_j}\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}} + \frac{\partial \hat{\boldsymbol{\beta}}'_{\boldsymbol{\lambda}}}{\partial \rho_i}\boldsymbol{H}_{\boldsymbol{\lambda}}\frac{\partial \hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}}{\partial \rho_j} + \frac{\partial \hat{\boldsymbol{\beta}}'_{\boldsymbol{\lambda}}}{\partial \rho_j}\frac{\partial \boldsymbol{S}_{\boldsymbol{\lambda}}}{\partial \rho_i}\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}\Big) + \hat{\boldsymbol{\beta}}'_{\boldsymbol{\lambda}}\frac{\partial^2 \boldsymbol{S}_{\boldsymbol{\lambda}}}{\partial \rho_i \partial \rho_j}\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}.
\end{aligned}$$

Using (4.4), the 1st and 2nd derivatives of $\boldsymbol{S}_{\boldsymbol{\lambda}}$ w.r.t. $\boldsymbol{\rho}$ can be substituted. It turns out that by defining $\boldsymbol{A}_{\boldsymbol{i}} = \exp(\rho_i)\tilde{\boldsymbol{S}}_{\boldsymbol{i}}\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}$ and $\boldsymbol{J}_{\boldsymbol{i}} = \partial\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}/\partial\rho_i$, the results are

$$\begin{aligned}
\frac{\partial D_p}{\partial \rho_i} &= \hat{\boldsymbol{\beta}}'_{\boldsymbol{\lambda}}\boldsymbol{A}_{\boldsymbol{i}} = \boldsymbol{A}'_{\boldsymbol{i}}\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}, \\
\frac{\partial^2 D_p}{\partial \rho_i \partial \rho_j} &= 2\big(\boldsymbol{J}'_{\boldsymbol{i}}\boldsymbol{A}_{\boldsymbol{j}} + \boldsymbol{J}'_{\boldsymbol{i}}\boldsymbol{H}_{\boldsymbol{\lambda}}\boldsymbol{J}_{\boldsymbol{j}} + \boldsymbol{J}'_{\boldsymbol{j}}\boldsymbol{A}_{\boldsymbol{i}}\big) + \delta_i^j\boldsymbol{A}'_{\boldsymbol{i}}\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}.
\end{aligned} \tag{4.7}$$

Assume that there are in total q smoothing parameters in $\boldsymbol{S}_{\boldsymbol{\lambda}}$. If we combine all $\boldsymbol{J}_{\boldsymbol{i}}$ and $\boldsymbol{A}_{\boldsymbol{i}}$ respectively into a matrix $\boldsymbol{J} = \begin{pmatrix}\boldsymbol{J_1} & \boldsymbol{J_2} & \cdots & \boldsymbol{J_q}\end{pmatrix}$ (this is a Jacobian matrix) and $\boldsymbol{A} = \begin{pmatrix}\boldsymbol{A_1} & \boldsymbol{A_2} & \cdots & \boldsymbol{A_q}\end{pmatrix}$, the derivatives can be written in a neat matrix form

$$\frac{\partial D_p}{\partial \boldsymbol{\rho}} = \boldsymbol{A}'\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}, \qquad \frac{\partial^2 D_p}{\partial \boldsymbol{\rho}\partial \boldsymbol{\rho}'} = 2(\boldsymbol{J}'\boldsymbol{A} + \boldsymbol{J}'\boldsymbol{H}_{\boldsymbol{\lambda}}\boldsymbol{J} + \boldsymbol{A}'\boldsymbol{J}) + \mathtt{diag}(\boldsymbol{A}'\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}).$$

Note however that $\boldsymbol{J}_{\boldsymbol{i}}$ is unknown yet. To derive it, we apply implicit differentiation. From (1.8) we have $\boldsymbol{H}_{\boldsymbol{\lambda}}\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}} = \boldsymbol{X}'\boldsymbol{W}\boldsymbol{y}$ where the RHS is independent of $\boldsymbol{\lambda}$. Differentiating both sides w.r.t. ρ_i gives

$$\frac{\partial \boldsymbol{H}_{\boldsymbol{\lambda}}}{\partial \rho_i}\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}} + \boldsymbol{H}_{\boldsymbol{\lambda}}\boldsymbol{J}_{\boldsymbol{i}} = \boldsymbol{0} \quad \Rightarrow \quad \boldsymbol{J}_{\boldsymbol{i}} = -\boldsymbol{V}_{\boldsymbol{\lambda}}\frac{\partial \boldsymbol{H}_{\boldsymbol{\lambda}}}{\partial \rho_i}\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}.$$

Furthermore, (1.7) implies (as $\boldsymbol{X}'\boldsymbol{W}\boldsymbol{X}$ is independent of $\boldsymbol{\lambda}$)

$$\frac{\partial \boldsymbol{H}_{\boldsymbol{\lambda}}}{\partial \rho_i} = \frac{\partial \boldsymbol{S}_{\boldsymbol{\lambda}}}{\partial \rho_i} \quad \Rightarrow \quad \boldsymbol{J}_{\boldsymbol{i}} = -\boldsymbol{V}_{\boldsymbol{\lambda}}\frac{\partial \boldsymbol{S}_{\boldsymbol{\lambda}}}{\partial \rho_i}\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},$$

Using (4.4) again, we get $\boldsymbol{J}_{\boldsymbol{i}} = -\boldsymbol{V}_{\boldsymbol{\lambda}}\boldsymbol{A}_{\boldsymbol{i}}$.

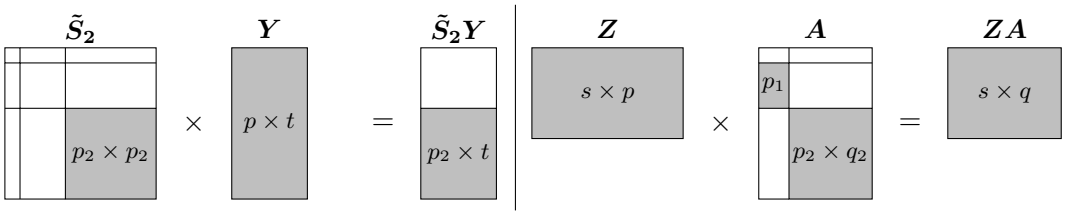


Figure 4.10: Two kinds of sparse matrix multiplications in $\boxed{D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda})}$ step and $\boxed{\log|\boldsymbol{H}_{\boldsymbol{\lambda}}|}$ step. In the graphs, dense blocks of a matrix are gray shaded, with its dimension printed at the centre. Unshaded blocks only contain 0. The $p \times p$ block diagonal penalty matrix $\tilde{\boldsymbol{S}}_{\boldsymbol{2}}$ is an example of $\tilde{\boldsymbol{S}}_{\boldsymbol{i}}$ (see the end of §4.3.1 if you forgot what it is). The left panel illustrates matrix multiplication $\tilde{\boldsymbol{S}}_{\boldsymbol{2}}\boldsymbol{Y}$, where $\boldsymbol{Y}$ is an arbitrary $p \times t$ dense matrix (if $t = 1$ then it is a vector). The resulting $p \times t$ matrix is also sparse, with a $p_2 \times t$ dense block. This multiplication involves $2p_2^2 t$ FLOP. The right panel illustrates matrix multiplication $\boldsymbol{Z}\boldsymbol{A}$, where $\boldsymbol{Z}$ is an arbitrary $s \times p$ dense matrix, and the $p \times q$ sparse matrix $\boldsymbol{A}$ is the one that is computed in $\boxed{D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda})}$ step. The resulting $s \times q$ matrix is dense, and the multiplication involves $2s(p_1 + q_2 p_2)$ FLOP.

At this point we are ready to see some very surprising result. By substituting $\boldsymbol{J}$ in $\partial^2 D_p/\partial\boldsymbol{\rho}\partial\boldsymbol{\rho}'$ with $\boldsymbol{J} = -\boldsymbol{V}_{\boldsymbol{\lambda}}\boldsymbol{A}$, we find that $\boldsymbol{J}'\boldsymbol{H}_{\boldsymbol{\lambda}}\boldsymbol{J} = \boldsymbol{A}'\boldsymbol{V}_{\boldsymbol{\lambda}}\boldsymbol{H}_{\boldsymbol{\lambda}}\boldsymbol{V}_{\boldsymbol{\lambda}}\boldsymbol{A} = \boldsymbol{A}'\boldsymbol{V}_{\boldsymbol{\lambda}}\boldsymbol{A}$ and $\boldsymbol{J}'\boldsymbol{A} = \boldsymbol{A}'\boldsymbol{J} = -\boldsymbol{A}'\boldsymbol{V}_{\boldsymbol{\lambda}}\boldsymbol{A}$, so that the first two terms in the bracket cancel out! In the end, there are simply

$$\frac{\partial D_p}{\partial \boldsymbol{\rho}} = \boldsymbol{A}'\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}, \quad \frac{\partial^2 D_p}{\partial \boldsymbol{\rho}\partial \boldsymbol{\rho}'} = 2\boldsymbol{J}'\boldsymbol{A} + \mathtt{diag}(\boldsymbol{A}'\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}). \tag{4.8}$$

In practice, computations of $\boldsymbol{A}$, $\boldsymbol{J}$ and $\boldsymbol{J}'\boldsymbol{A}$ can exploit sparsity and are very efficient. Figure 4.10 illustrates two kinds of matrix multiplications involving sparse matrices. In particular, $\boldsymbol{A}$ is computed column by column, where computation of column $\boldsymbol{A}_{\boldsymbol{i}}$ belongs to the first knid. Computations of $\boldsymbol{J}$ and $\boldsymbol{J}'\boldsymbol{A}$ belong to the second kind. For the example additive model considered in this section, these computations respectively involve $2q_2p_2^2$, $2p(p_1 + q_2p_2)$ and $2q(p_1 + q_2p_2)$ FLOP.

4.3.6 Derivatives of $\log|\boldsymbol{H}_{\boldsymbol{\lambda}}|$

Using Harville (1997, equation (8.6), §15.8; equation (9.3), §15.9) again, it can be derived that

$$\frac{\partial \log|\boldsymbol{H}_{\boldsymbol{\lambda}}|}{\partial \rho_i} = \exp(\rho_i)\mathtt{tr}\big(\boldsymbol{V}_{\boldsymbol{\lambda}}\tilde{\boldsymbol{S}}_{\boldsymbol{i}}\big),$$
$$\frac{\partial^2 \log|\boldsymbol{H}_{\boldsymbol{\lambda}}|}{\partial \rho_i \partial \rho_j} = \delta_i^j \exp(\rho_i)\mathtt{tr}\big(\boldsymbol{V}_{\boldsymbol{\lambda}}\tilde{\boldsymbol{S}}_{\boldsymbol{i}}\big) - \exp(\rho_i + \rho_j)\mathtt{tr}\big(\boldsymbol{V}_{\boldsymbol{\lambda}}\tilde{\boldsymbol{S}}_{\boldsymbol{i}}\boldsymbol{V}_{\boldsymbol{\lambda}}\tilde{\boldsymbol{S}}_{\boldsymbol{j}}\big),$$

where the last trace also equals $\mathtt{tr}\big(\tilde{\boldsymbol{S}}_{\boldsymbol{i}}\boldsymbol{V}_{\boldsymbol{\lambda}}\tilde{\boldsymbol{S}}_{\boldsymbol{j}}\boldsymbol{V}_{\boldsymbol{\lambda}}\big)$. Define $\boldsymbol{C}_{\boldsymbol{i}} = \tilde{\boldsymbol{S}}_{\boldsymbol{i}}\boldsymbol{V}_{\boldsymbol{\lambda}}$, then there are

$$\frac{\partial \log|\boldsymbol{H}_{\boldsymbol{\lambda}}|}{\partial \rho_i} = \exp(\rho_i)\mathtt{tr}(\boldsymbol{C}_{\boldsymbol{i}}),$$
$$\frac{\partial^2 \log|\boldsymbol{H}_{\boldsymbol{\lambda}}|}{\partial \rho_i \partial \rho_j} = \delta_i^j \exp(\rho_i)\mathtt{tr}(\boldsymbol{C}_{\boldsymbol{i}}) - \exp(\rho_i + \rho_j)\mathtt{tr}(\boldsymbol{C}_{\boldsymbol{i}}\boldsymbol{C}_{\boldsymbol{j}}).$$

Computation of $\boldsymbol{C}_{\boldsymbol{i}}$ belongs to the first type of matrix multiplications illustrated in Figure 4.10. As a result, $\boldsymbol{C}_{\boldsymbol{i}}$ is a $p \times p$ sparse matrix which only has one chunk of non-zero rows. Suppose these rows are indexed by a vector $\boldsymbol{k}_{\boldsymbol{i}}$, then under the initial reparametrization, there are $\boldsymbol{C}_{\boldsymbol{i}}(\boldsymbol{k}_{\boldsymbol{i}},\ ) = \boldsymbol{V}_{\boldsymbol{\lambda}}(\boldsymbol{k}_{\boldsymbol{i}},\ )$ for a single-λ block, and $\boldsymbol{C}_{\boldsymbol{i}}(\boldsymbol{k}_{\boldsymbol{i}},\ ) = \bar{\boldsymbol{S}}_{\boldsymbol{i}}\boldsymbol{V}_{\boldsymbol{\lambda}}(\boldsymbol{k}_{\boldsymbol{i}},\ )$ for a multi-λ block. Clearly for a single-λ block, $\boldsymbol{C}_{\boldsymbol{i}}$ needs not be computed; nor does it needs be stored as we can simply store the row index vector $\boldsymbol{k}_{\boldsymbol{i}}$. Thus, computation of $\boldsymbol{C}_{\boldsymbol{i}}$ has a complexity that depends on the size of multi-λ blocks only. For the example additive model considered in this section, this involves $2q_2p_2^2p$ FLOP. Subsequent computations of $\mathtt{tr}(\boldsymbol{C}_{\boldsymbol{i}})$ and $\mathtt{tr}(\boldsymbol{C}_{\boldsymbol{i}}\boldsymbol{C}_{\boldsymbol{j}})$ are very efficient by exploiting sparsity. For example, Figure 4.11 illustrates the computation of the latter.

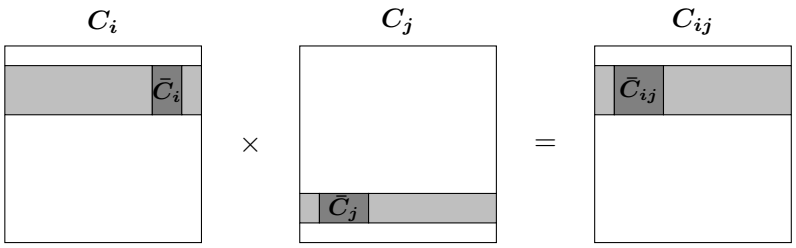


Figure 4.11: Computation of $\mathtt{tr}(\boldsymbol{C_iC_j})$ in $\boxed{\log|\boldsymbol{H_\lambda}|}$ step. In the matrix product $\boldsymbol{C_{ij}} = \boldsymbol{C_iC_j}$, only the chunk of rows indexed by $\boldsymbol{k_i}$ is non-zero. Hence, there is $\mathtt{tr}(\boldsymbol{C_{ij}}) = \mathtt{tr}(\bar{\boldsymbol{C}}_{\boldsymbol{ij}}) = \mathtt{tr}(\bar{\boldsymbol{C}}_{\boldsymbol{i}}\bar{\boldsymbol{C}}_{\boldsymbol{j}})$, where $\bar{\boldsymbol{C}}_{\boldsymbol{ij}} = \boldsymbol{C_{ij}}(\boldsymbol{k_i},\ \boldsymbol{k_i})$, $\bar{\boldsymbol{C}}_{\boldsymbol{i}} = \boldsymbol{C_i}(\boldsymbol{k_i},\ \boldsymbol{k_j})$, and $\bar{\boldsymbol{C}}_{\boldsymbol{j}} = \boldsymbol{C_j}(\boldsymbol{k_j},\ \boldsymbol{k_i})$. Computation of $\mathtt{tr}(\bar{\boldsymbol{C}}_{\boldsymbol{i}}\bar{\boldsymbol{C}}_{\boldsymbol{j}})$ needs no matrix multiplication. Similar to the computation of $\mathtt{tr}(\boldsymbol{D_{2i}D_{2j}})$ in $\boxed{\log|\boldsymbol{S_\lambda}|_+}$ step, it is just the sum of an element-wise product between $\bar{\boldsymbol{C}}_{\boldsymbol{i}}$ and $\bar{\boldsymbol{C}}'_{\boldsymbol{j}}$.

4.3.7 Special numerical issue: evaluation of $\log|\boldsymbol{S_\lambda}|_+$

This section comes back to the evaluation of $\log|\bar{\boldsymbol{S}}_{\boldsymbol{2}}| = \log|\sum_{i=1}^{q_2}\lambda_{2i}\bar{\boldsymbol{S}}_{\boldsymbol{2i}}|$ in the example additive model, or in general the log-determinant of a multi-λ block $\boldsymbol{M} = \sum_{i=1}^{q}\lambda_i\boldsymbol{M_i}$, where

- λ_i's are positive real values;
- $\boldsymbol{M_i}$'s are positive-semidefinite matrics that do not have full rank;
- $\boldsymbol{M}$ has full rank.

In §4.3.2 I have explained its basic computation methods via a pivoted QR factorization or a pivoted Cholesky factorization, with a pre-notice that it does not always give the correct answer. Now I am going to fully elaborate this matter in the following steps.

1. Present a small example where direct numerical evaluation can indeed be incorrect;
2. Explain why direct numerical evaluation goes wrong for the example;
3. Provide a way to overcome this issue and formulate this problem in general.

Let us start with the following example with $q = 2$ penalty matrices $\boldsymbol{M} = \lambda_1\boldsymbol{M_1} + \lambda_2\boldsymbol{M_2}$:

$$\boldsymbol{M_1} = \begin{pmatrix} 1 & -1 & \cdot & \cdot & \cdot \\ -1 & 2 & -1 & \cdot & \cdot \\ \cdot & -1 & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix}, \quad \boldsymbol{M_2} = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 \end{pmatrix},$$

where zero elements in the matrices are marked by '$\cdot$'. Let $\tau = \lambda_1/\lambda_2$, the log-determinant equals to

$$\log|\lambda_2(\tau\boldsymbol{M_1} + \boldsymbol{M_2})| = \log\big(|\lambda_2\boldsymbol{I}|\cdot|\tau\boldsymbol{M_1} + \boldsymbol{M_2}|\big) = 5\log\lambda_2 + \log|\tau\boldsymbol{M_1} + \boldsymbol{M_2}|,$$

so let us focus on computing $\log|\tau\boldsymbol{M_1} + \boldsymbol{M_2}|$. Analytical result can be easily derived:

$$|\tau\boldsymbol{M_1} + \boldsymbol{M_2}| = \begin{vmatrix} \tau & -\tau & \cdot & \cdot & \cdot \\ -\tau & 2\tau & -\tau & \cdot & \cdot \\ \cdot & -\tau & \tau+1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 \end{vmatrix} = \tau^2 \quad \Rightarrow \quad \log|\tau\boldsymbol{M_1} + \boldsymbol{M_2}| = 2\log\tau.$$

However, Table 4.2 shows that while numerical computation of this log-determinant is still correct as $\tau \to 0$, it becomes badly wrong as $\tau \to +\infty$. Particularly, for $\tau = 10^{16}$ the Cholesky method returns

Table 4.2: Numerical computation of $\log|\tau\boldsymbol{M_1}+\boldsymbol{M_2}|$ can be wrong as $\tau\to+\infty$. See main text for $\boldsymbol{M_1}$, $\boldsymbol{M_2}$ and analytical value of the log-determinant used for this experiment.

τ	1	10^{-4}	10^{-8}	10^{-12}	10^{-16}	10^{-20}
Analytical	0.00000	-18.42068	-36.84136	-55.26204	-73.68272	-92.10340
QR	0.00000	-18.42068	-36.84136	-55.26204	-73.68272	-92.10340
Cholesky	0.00000	-18.42068	-36.84136	-55.26204	-73.68272	-92.10340

τ	1	10^{4}	10^{8}	10^{12}	10^{16}	10^{20}
Analytical	0.00000	18.42068	36.84136	55.26204	73.68272	92.10340
QR	0.00000	18.42068	36.84136	55.26254	75.33064	102.35677
Cholesky	0.00000	18.42068	36.84136	55.26146	-Inf	102.11145

$-\infty$. This implies that $\boldsymbol{M}=\tau\boldsymbol{M_1}+\boldsymbol{M_2}$ becomes numerically rank-deficient, and some diagonal elements of its Cholesky factor are zeros.

To understand this issue, consider an eigen decomposition (see §4.1.5 if you need a revision) $\boldsymbol{M_1}=\boldsymbol{UDU'}$ and a similarity transformation $\boldsymbol{E}=\boldsymbol{U'M_2U}$, then there is $\log|\tau\boldsymbol{M_1}+\boldsymbol{M_2}|=\log|\tau\boldsymbol{D}+\boldsymbol{E}|$. We first show analytical results. Denote $\dot{\boldsymbol{M}}_{\boldsymbol{1}}=\boldsymbol{M_1}(1:3,\,1:3)$. Its two eigenvalues $\omega_1=3$ and $\omega_2=1$ can be easily obtained by solving the characteristic equation $|\dot{\boldsymbol{M}}_{\boldsymbol{1}}-\omega\boldsymbol{I}|=\omega(1-\omega)(\omega-3)=0$. It is then straightforward to obtain its eigenvectors (scaled to have unit L_2-norm) $\boldsymbol{\nu_1}=(\sqrt{6}/6,\,-\sqrt{6}/3,\,\sqrt{6}/6)'$ and $\boldsymbol{\nu_2}=(-\sqrt{2}/2,\,0,\,\sqrt{2}/2)'$ by solving linear equations $(\dot{\boldsymbol{S}}_{\boldsymbol{1}}-\boldsymbol{I})\boldsymbol{\nu_1}=\boldsymbol{0}$ and $(\dot{\boldsymbol{S}}_{\boldsymbol{1}}-3\boldsymbol{I})\boldsymbol{\nu_2}=\boldsymbol{0}$. It follows that the eigen decomposition can be writen as

$$\boldsymbol{D}=\begin{pmatrix}3&\cdot&\cdot&\cdot&\cdot\\ \cdot&1&\cdot&\cdot&\cdot\\ \cdot&\cdot&\cdot&\cdot&\cdot\\ \cdot&\cdot&\cdot&\cdot&\cdot\\ \cdot&\cdot&\cdot&\cdot&\cdot\end{pmatrix},\quad \boldsymbol{U}=\begin{pmatrix}\sqrt{6}/6&-\sqrt{2}/2&\sqrt{3}/3&\cdot&\cdot\\ -\sqrt{6}/3&\cdot&\sqrt{3}/3&\cdot&\cdot\\ \sqrt{6}/6&\sqrt{2}/2&\sqrt{3}/3&\cdot&\cdot\\ \cdot&\cdot&\cdot&1&\cdot\\ \cdot&\cdot&\cdot&\cdot&1\end{pmatrix},$$

where $\boldsymbol{U}(,\,1)=\boldsymbol{\nu_1}$ and $\boldsymbol{U}(,\,2)=\boldsymbol{\nu_2}$. Other columns of $\boldsymbol{U}$ can be determined as they must have unit L_2-norm and be orthogonal to $\boldsymbol{\nu_1}$ and $\boldsymbol{\nu_2}$ (so that $\boldsymbol{U}$ is an orthonormal matrix). The similarity transformation is thus

$$\boldsymbol{E}=\boldsymbol{U'M_2U}=\begin{pmatrix}1/6&\sqrt{3}/6&\sqrt{2}/6&\cdot&\cdot\\ \sqrt{3}/6&1/2&\sqrt{6}/6&\cdot&\cdot\\ \sqrt{2}/6&\sqrt{6}/6&1/3&\cdot&\cdot\\ \cdot&\cdot&\cdot&1&\cdot\\ \cdot&\cdot&\cdot&\cdot&1\end{pmatrix},$$

and the determinant is

$$|\tau\boldsymbol{D}+\boldsymbol{E}|=\begin{vmatrix}3\tau+1/6&\sqrt{3}/6&\sqrt{2}/6&\cdot&\cdot\\ \sqrt{3}/6&\tau+1/2&\sqrt{6}/6&\cdot&\cdot\\ \sqrt{2}/6&\sqrt{6}/6&1/3&\cdot&\cdot\\ \cdot&\cdot&\cdot&1&\cdot\\ \cdot&\cdot&\cdot&\cdot&1\end{vmatrix}=\tau^2.$$

When it comes to numerical computation, the computed eigen decomposition is

$$\hat{\boldsymbol{D}}=\begin{pmatrix}3&\cdot&\cdot&\cdot&\cdot\\ \cdot&1&\cdot&\cdot&\cdot\\ \cdot&\cdot&2.66\times10^{-15}&\cdot&\cdot\\ \cdot&\cdot&\cdot&\cdot&\cdot\\ \cdot&\cdot&\cdot&\cdot&\cdot\end{pmatrix},\quad \hat{\boldsymbol{U}}=\begin{pmatrix}0.40825&-0.70711&0.57735&\cdot&\cdot\\ -0.81650&\cdot&0.57735&\cdot&\cdot\\ 0.40825&0.70711&0.57735&\cdot&\cdot\\ \cdot&\cdot&\cdot&1&\cdot\\ \cdot&\cdot&\cdot&\cdot&1\end{pmatrix},$$

Table 4.3: Numerical computation of $\log|\tau\boldsymbol{S_1}+\boldsymbol{S_2}|$ becomes correct as $\tau\to+\infty$, if we first compute an eigen decomposition $\boldsymbol{S_1}=\hat{\boldsymbol{U}}\hat{\boldsymbol{D}}\hat{\boldsymbol{U}}'$, detect the numerical rank of $\boldsymbol{S_1}$, set the spurious eigenvalues in $\hat{\boldsymbol{D}}$ to zeros, then compute $\log|\tau\hat{\boldsymbol{D}}+\hat{\boldsymbol{U}}'\boldsymbol{S_2}\hat{\boldsymbol{U}}|$ following a similarity transformation.

τ	1	10^4	10^8	10^{12}	10^{16}	10^{20}
Analytical	0.00000	18.42068	36.84136	55.26204	73.68272	92.10340
QR	0.00000	18.42068	36.84136	55.26204	73.68272	92.10340
Cholesky	0.00000	18.42068	36.84136	55.26204	73.68272	92.10340

and the computed similarity transformation is

$$\hat{\boldsymbol{E}}=\begin{pmatrix}0.16667 & 0.28868 & 0.23570 & \cdot & \cdot\\ 0.28868 & 0.50000 & 0.40825 & \cdot & \cdot\\ 0.23570 & 0.40825 & 0.33333 & \cdot & \cdot\\ \cdot & \cdot & \cdot & 1 & \cdot\\ \cdot & \cdot & \cdot & \cdot & 1\end{pmatrix}.$$

We see that although $\hat{\boldsymbol{U}}$ and $\hat{\boldsymbol{E}}$ are identical to $\boldsymbol{U}$ and $\boldsymbol{E}$, the third computed eigenvalue, denoted by δ, is not strictly zero, but a small value at the magnitude of the machine precision (about 2.22×10^{-16}). As a result, we get

$$|\tau\hat{\boldsymbol{D}}+\hat{\boldsymbol{E}}|=\begin{vmatrix}3\tau+1/6 & \sqrt{3}/6 & \sqrt{2}/6 & \cdot & \cdot\\ \sqrt{3}/6 & \tau+1/2 & \sqrt{6}/6 & \cdot & \cdot\\ \sqrt{2}/6 & \sqrt{6}/6 & \delta\tau+1/3 & \cdot & \cdot\\ \cdot & \cdot & \cdot & 1 & \cdot\\ \cdot & \cdot & \cdot & \cdot & 1\end{vmatrix}=\tau^2+\delta\tau\big(3\tau^2+\frac{5}{3}\tau+\frac{1}{36}\big),$$

which deviates from the true quantity by $\delta\tau(3\tau^2+\frac{5}{3}\tau+\frac{1}{36})$. As $\tau\to+\infty$, this error is toward $+\infty$.

The above demonstration also hints a way to suppress such undesired numerical property. If we, after the computed eigen decomposition, manually set $\delta=0$, then $|\tau\hat{\boldsymbol{D}}+\hat{\boldsymbol{E}}|$ would give the correct result. Table 4.3 verifies this.

Suppose that $\boldsymbol{S_1}$ has rank $r<t$, then $\tau\hat{\boldsymbol{D}}+\hat{\boldsymbol{E}}$ can be partitioned as

$$\begin{pmatrix}\tau\hat{\boldsymbol{D}}_{\boldsymbol{11}}+\hat{\boldsymbol{E}}_{\boldsymbol{11}} & \hat{\boldsymbol{E}}_{\boldsymbol{12}}\\ \hat{\boldsymbol{E}}_{\boldsymbol{21}} & \tau\hat{\boldsymbol{D}}_{\boldsymbol{22}}+\hat{\boldsymbol{E}}_{\boldsymbol{22}}\end{pmatrix},$$

where $\hat{\boldsymbol{D}}_{\boldsymbol{11}}=\hat{\boldsymbol{D}}(1:r,\ 1:r)$ contains the correct eigenvalues of $\boldsymbol{S_1}$ while $\hat{\boldsymbol{D}}_{\boldsymbol{22}}$ contains spurious eigenvalues with $|\hat{\boldsymbol{D}}_{\boldsymbol{22}}(i,\ i)|<\hat{\boldsymbol{D}}_{\boldsymbol{11}}(1,\ 1)\epsilon$. Since $\tau\hat{\boldsymbol{D}}_{\boldsymbol{11}}+\hat{\boldsymbol{E}}_{\boldsymbol{11}}$ has full rank, then using the general result

$$\begin{vmatrix}\boldsymbol{A_{11}} & \boldsymbol{A_{12}}\\ \boldsymbol{A_{21}} & \boldsymbol{A_{22}}\end{vmatrix}=|\boldsymbol{A_{11}}|\cdot|\boldsymbol{A_{22}}-\boldsymbol{A_{21}}\boldsymbol{A_{11}^{-1}}\boldsymbol{A_{12}}|,$$

there is

$$\log|\tau\hat{\boldsymbol{D}}+\hat{\boldsymbol{E}}|=\log|\tau\hat{\boldsymbol{D}}_{\boldsymbol{11}}+\hat{\boldsymbol{E}}_{\boldsymbol{11}}|+\log|\tau\hat{\boldsymbol{D}}_{\boldsymbol{22}}+\hat{\boldsymbol{E}}_{\boldsymbol{22}}-\hat{\boldsymbol{E}}_{\boldsymbol{21}}(\tau\hat{\boldsymbol{D}}_{\boldsymbol{11}}+\hat{\boldsymbol{E}}_{\boldsymbol{11}})^{-1}\hat{\boldsymbol{E}}_{\boldsymbol{12}}|.$$

As $\tau\to+\infty$, the last log-determinant would approach $\log|\tau\hat{\boldsymbol{D}}_{\boldsymbol{22}}+\hat{\boldsymbol{E}}_{\boldsymbol{22}}|$, which would only be correct if we set $\hat{\boldsymbol{D}}_{\boldsymbol{22}}=\boldsymbol{0}$, which we call a "truncation". Otherwise, an upper bound of the wrong value is

$$\log|\tau\hat{\boldsymbol{D}}_{\boldsymbol{22}}+\hat{\boldsymbol{E}}_{\boldsymbol{22}}|<\log|\tau\hat{\boldsymbol{D}}_{\boldsymbol{11}}(1,\ 1)\epsilon\boldsymbol{I}+\hat{\boldsymbol{E}}_{\boldsymbol{22}}|=\textstyle\sum_i\log\big(\tau\hat{\boldsymbol{D}}_{\boldsymbol{11}}(1,\ 1)\epsilon+e_i\big),$$

where e_i is $\hat{\boldsymbol{E}}_{\boldsymbol{22}}$'s eigenvalue, which is also $\hat{\boldsymbol{E}}$'s and hence $\boldsymbol{S_2}$'s eigenvalue. An upper bound for the error is then $\sum_i\log\big(\tau\hat{\boldsymbol{D}}_{\boldsymbol{11}}(1,\ 1)\epsilon/e_i+1\big)$. Obviously, whenever the maximum eigenvalue of $\boldsymbol{S_1}$ times $\tau\epsilon$ is much larger than the smallest eigenvalue of $\boldsymbol{S_2}$, a situation we call a "bad scaling" between $\boldsymbol{S_1}$ and $\boldsymbol{S_2}$, the error bound is non-negligible. This may give some heuristic on when a similarity transformation and a "truncation" are possibly needed for stable evaluation of a log-determinant. A more-than-sufficient condition is as such. Compute Frobenius norm (an upper bound for maximum

Table 4.4: FLOP count for the fives computation steps in computing $\boldsymbol{\nabla}\mathcal{V}_r$ and $\boldsymbol{\nabla^2}\mathcal{V}_r$. See main text for meanings of l, p_i, q_i, p and q.

$\boxed{\log\lvert\boldsymbol{S}_{\boldsymbol{\lambda}}\rvert_+}$	$\boxed{\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}}$	$\boxed{\boldsymbol{V}_{\boldsymbol{\lambda}}}$	$\boxed{D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda})}$	$\boxed{\log\lvert\boldsymbol{H}_{\boldsymbol{\lambda}}\rvert}$
$\sum_{\{i:\, q_i>1\}} (3q_i + \frac{4}{3})p_i^3$	$\frac{10}{3}p^3$	$\frac{2}{3}p^3$	$\sum_{\{i:\, q_i>1\}} 2q_ip_i^2 + 2(p+q)\sum_{i=1}^{l} q_ip_i$	$\sum_{\{i:\, q_i>1\}} 2q_ip_i^2p$

eigenvalue) $\|\boldsymbol{S_1}\|_F$ and $\|\boldsymbol{S_2}\|_F$ of $\boldsymbol{S_1}$ and $\boldsymbol{S_2}$. If $\lambda_1\|\boldsymbol{S_1}\|_F/\lambda_2\|\boldsymbol{S_2}\|_F > \epsilon$, then perform similarity transformation and "truncation".

If there are $q > 2$ penalty matrices in a multi-λ block, the above idea for $q = 2$ case can be generalized. We refer readers to Wood (2011, Appendix B) for such an algorithm. In the end, there exists a single similarity transformation $\dot{\boldsymbol{S}} = \dot{\boldsymbol{U}}'\boldsymbol{S}\dot{\boldsymbol{U}}$, and evaluation of $\log|\dot{\boldsymbol{S}}|$ is numerically stable. Such transformation also guarantees that $\dot{\boldsymbol{S}}^{-1}$ is computable for calculations of derivatives. Note that a transformation to each individual penalty matrix $\dot{\boldsymbol{S}}_{\boldsymbol{i}} = \dot{\boldsymbol{U}}'\boldsymbol{S_i}\dot{\boldsymbol{U}}$ is needed. In addition, in the subsequent $\boxed{\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}}$ step, $\boldsymbol{R}^*$ needs be transformed accordingly.

4.3.8 Summary

In §4.3.2 to §4.3.6 I have given FLOP count of the five computation steps in computing $\boldsymbol{\nabla}\mathcal{V}_r$ and $\boldsymbol{\nabla^2}\mathcal{V}_r$ for the example additive model in §4.3.1. Now I will generalize these results. Suppose we have an additive model with $\boldsymbol{X\beta} = \boldsymbol{X_0\beta_0} + \sum_{i=1}^{l}\boldsymbol{f_i}$, where there are some parametric term $\boldsymbol{\beta_0}$ and l splines. For the i^{th} spline $\boldsymbol{f_i} = \boldsymbol{X_i\beta_i}$, assume it has p_i coeffcients in $\boldsymbol{\beta_i}$ and q_i smoothing parameters in its penalty matrix $\boldsymbol{S_i}$. Note that q_i may be any non-negative integers. If $q_i = 0$, then $\boldsymbol{f_i}$ is a pure regression spline without penalization. If $q_i = 1$, then $\boldsymbol{S_i}$ is a single-λ block. $q_i > 1$ implies that $\boldsymbol{S_i}$ is a multi-λ block. The total number of coefficients in $\boldsymbol{\beta}$ is $p = \sum_{i=0}^{l} p_i$ (don't forget the parametric term) and the total number of smoothing parameters in $\boldsymbol{S_\lambda}$ is $q = \sum_{i=1}^{l} q_i$. For this general additive model, computational complexity of all steps is summarized in Table 4.4. Results for $\boxed{\log|\boldsymbol{S}_{\boldsymbol{\lambda}}|_+}$ and $\boxed{\log|\boldsymbol{H}_{\boldsymbol{\lambda}}|}$ steps only depend on the size of multi-λ blocks in $\boldsymbol{S_\lambda}$, as the summation is over $\{i : q_i > 1\}$. In addition, the result for $\boxed{\log|\boldsymbol{S}_{\boldsymbol{\lambda}}|_+}$ step is for QR factorization approach; for Cholesky factorization approach, the sequence to be summed is $(2q_i + \frac{1}{3})p_i^3$. In general, $\boxed{\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}}$ step is the most expensive step, and its $O(p^3)$ complexity is often reported as an approximate complexity for REML estimation, in contrast to the $O(np^2)$ complexity for model matrix reduction.

Apart from parameter estimation, prediction of response variable $\boldsymbol{y}$, i.e., computation of $\hat{\boldsymbol{y}} = \boldsymbol{X}\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}$, is also an important part of inference. This is a straightforward matrix-vector multiplication, but to avoid forming the whole $\boldsymbol{X}$, the row chunking strategy used in model matrix reduction (see §4.2 if you need a revision) can be likewise applied, with further possibility of parallel computing. In addition, prediction requires the original parametrization of the coefficient vector. However, the initial reparametrization before REML estimation means that at convergence of the Newton-Raphson algorithm, estimated coefficients and unscaled covariance matrix are actually associated with $\tilde{\boldsymbol{\beta}}$ not $\boldsymbol{\beta}$. Denote these estimates by $\tilde{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}$ and $\tilde{\boldsymbol{V}}_{\boldsymbol{\lambda}}$, back transformations $\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}} = \boldsymbol{U}^{-1}\tilde{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}$ and $\boldsymbol{V_\lambda} = \boldsymbol{U}^{-1}\tilde{\boldsymbol{V}}_{\boldsymbol{\lambda}}\boldsymbol{U}'^{-1}$ are a necessary post-processing. In case a secondary reparametrization for stable evaluation of $\log|\boldsymbol{S_\lambda}|_+$ also exists, it needs be reversed in a similar way and should be done in the first place.

4.4 Estimation of GAMs via "performance iteration"

A GAM can be expressed as a penalized generalized linear model (GLM), or a generalized linear mixed model (GLMM), as follows:

$$\boldsymbol{y}\,|\,\boldsymbol{\beta} \sim \texttt{EF}(\boldsymbol{\mu},\ \phi), \qquad \boldsymbol{\eta} = g(\boldsymbol{\mu}) = \boldsymbol{X\beta}, \qquad \boldsymbol{\beta} \sim N(\boldsymbol{0},\ \boldsymbol{S}_{\boldsymbol{\lambda}}^{-1}\phi),$$

Initialize $k = 0$, $D_p^{[0]} = 0$, $\boldsymbol{\mu}^{[0]} = \boldsymbol{y} + \boldsymbol{\xi}$ and $\boldsymbol{\eta}^{[0]} = g(\boldsymbol{\mu}^{[0]})$, then

1. obtain pseudo data vector $\boldsymbol{z}^{[k]}$, with elements $z_i^{[k]} = g'(\mu_i^{[k]})(y_i - \mu_i^{[k]}) + \eta_i^{[k]}$, and a diagonal weight matrix $\boldsymbol{W}^{[k]}$, with diagonal elements $w_{ii}^{[k]} = V(\mu_i^{[k]})^{-1} g'(\mu_i^{[k]})^{-2}$;
2. solve a penalized weighted least squares problem $\boldsymbol{\beta}^{[k+1]} = \underset{\boldsymbol{\beta}}{\arg\max}\, D_p(\boldsymbol{\beta},\ \boldsymbol{\lambda})$, where $D_p(\boldsymbol{\beta},\ \boldsymbol{\lambda}) = \left\|(\boldsymbol{W}^{[k]})^{\frac{1}{2}}\boldsymbol{z}^{[k]} - (\boldsymbol{W}^{[k]})^{\frac{1}{2}}\boldsymbol{X}\boldsymbol{\beta}\right\|^2 + \boldsymbol{\beta}'\boldsymbol{S}_{\boldsymbol{\lambda}}\boldsymbol{\beta}$;
3. evaluate $D_p^{[k+1]} = D_p(\boldsymbol{\beta}^{[k+1]},\ \boldsymbol{\lambda})$, $\boldsymbol{\eta}^{[k+1]} = \boldsymbol{X}\boldsymbol{\beta}^{[k+1]}$ and $\boldsymbol{\mu}^{[k+1]} = g^{-1}\big(\boldsymbol{\eta}^{[k+1]}\big)$;
4. if $|D_p^{[k+1]} - D_p^{[k]}| < (0.1 + |D_p^{[k]}|)\epsilon$, then iteration has converged and go to 5; otherwise $k \mathrel{+}= 1$ and go to 1.
5. set $\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}} = \boldsymbol{\beta}^{[k+1]}$, $\hat{\boldsymbol{\eta}}_{\boldsymbol{\lambda}} = \boldsymbol{\eta}^{[k+1]}$, $\hat{\boldsymbol{\mu}}_{\boldsymbol{\lambda}} = \boldsymbol{\mu}^{[k+1]}$, $\boldsymbol{W}_{\boldsymbol{\lambda}} = \boldsymbol{W}^{[k]}$, $\boldsymbol{z}_{\boldsymbol{\lambda}} = \boldsymbol{z}^{[k]}$ and $D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda}) = \left\|\boldsymbol{W}_{\boldsymbol{\lambda}}^{\frac{1}{2}}\boldsymbol{z}_{\boldsymbol{\lambda}} - \boldsymbol{W}_{\boldsymbol{\lambda}}^{\frac{1}{2}}\boldsymbol{X}\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}\right\|^2 + \hat{\boldsymbol{\beta}}'_{\boldsymbol{\lambda}}\boldsymbol{S}_{\boldsymbol{\lambda}}\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}$.

Figure 4.12: P-IRLS algorithm for GAM estimation with known $\boldsymbol{\lambda}$. ϵ is some small numerical tolerance. $\boldsymbol{\xi}$ is a vector of small quantities (often zeros) to ensure that $g(\boldsymbol{\mu}^{[0]})$ exists. $g'(\mu)$ is the 1st derivative of the link function $g(\mu)$. $V(\mu)$ is the variance function for the exponential family distribution. Note that at convergence (step 5) of the algorithm, all quantities have dependency on $\boldsymbol{\lambda}$.

where `EF` is an exponential family distribution with known or unknown scale parameter ϕ, and g is a known smooth monotonic link function that transforms $\boldsymbol{\mu} = \mathtt{E}(\boldsymbol{y}|\boldsymbol{\beta})$ to the linear predictor $\boldsymbol{\eta}$. If all smoothing parameters in $\boldsymbol{\lambda}$ are known, then $\boldsymbol{\beta}$ can be estimated by maximizing the penalized log-likelihood (which in Bayesian point of view, differs from the log-posterior by some $\boldsymbol{\lambda}$-related constant):

$$l_p(\boldsymbol{\beta},\ \boldsymbol{\lambda}) = l(\boldsymbol{\beta}) - \frac{\boldsymbol{\beta}'\boldsymbol{S}_{\boldsymbol{\lambda}}\boldsymbol{\beta}}{2\phi}.$$

Numerical maximization often adopts a *penalized iteratively re-weighted least squares (P-IRLS)* algorithm, as presented in Figure 4.12.

When $\boldsymbol{\lambda}$ is unknown and also needs be estimated, step 2 of the P-IRLS algorithm may be replaced by estimation of an additive model

$$\boldsymbol{z}^{[k]}|\boldsymbol{\beta} \sim N(\boldsymbol{X}\boldsymbol{\beta},\ (\boldsymbol{W}^{[k]})^{-1}\phi), \qquad \boldsymbol{\beta} \sim N(\boldsymbol{0},\ \boldsymbol{S}_{\boldsymbol{\lambda}}^{-1}\phi),$$

using methods explained in previous sections of this Chapter. The resulting algorithm resembles the *penalized quasi-likelihood (PQL)* estimation (Breslow and Clayton, 1993; Tuerlinckx et al., 2010) of a GLMM. It may also be seen as a variant of the "performance iteration" proposed by Gu (1992) for GAM estimation using generalized cross-validation. The normality assumption for $\boldsymbol{z}^{[k]}$ is clearly false. However, after the QR reduction $(\boldsymbol{W}^{[k]})^{\frac{1}{2}}\boldsymbol{z}^{[k]} = \boldsymbol{Q}^{[k]\prime}\boldsymbol{R}^{*[k]}$, $D_p(\boldsymbol{\beta},\ \lambda)$ in step 2 of the P-IRLS algorithm can be rewritten into a form similar to (4.2), but now with $\boldsymbol{f}^{[k]} = \boldsymbol{Q}^{[k]}(\prime\boldsymbol{W}^{[k]})^{\frac{1}{2}}\boldsymbol{z}^{[k]}$. This implies that the additive model for $\boldsymbol{z}^{[k]}$ may be alternatively written as the following one for $\boldsymbol{f}^{[k]}$:

$$\boldsymbol{f}^{[k]}|\boldsymbol{\beta} \sim N(\boldsymbol{R}^{*[k]}\boldsymbol{\beta},\ \boldsymbol{I}\phi), \qquad \boldsymbol{\beta} \sim N(\boldsymbol{0},\ \boldsymbol{S}_{\boldsymbol{\lambda}}^{-1}\phi),$$

For large datasets there is $n \gg p$, so asymptotic normality for $\boldsymbol{f}^{[k]}$ is justified under central limit theorem.

"Performance" iteration is not the only method for GAM estimation. Alternatively, we may attempt a number of candidate smoothing parameters $\boldsymbol{\lambda}^{[1]}$, $\boldsymbol{\lambda}^{[2]}$, ..., and for each trial the GAM is estimated with the P-IRLS algorithm. Then we choose the best $\boldsymbol{\lambda}^{[i]}$, which minimizes the REML score $\mathcal{V}_r(\boldsymbol{\lambda},\ \phi) = -2\log\int \mathtt{P}(\boldsymbol{y}|\boldsymbol{\beta})\mathtt{P}(\boldsymbol{\beta})\mathrm{d}\boldsymbol{\beta}$. This crude "grid search" can be replaced by numerical minimization via a Newton-Raphson algorithm, if $\boldsymbol{\nabla}\mathcal{V}_r$ and $\boldsymbol{\nabla}^2\mathcal{V}_r$ can be derived. The resulting estimation method is termed the "outer iteration" in `mgcv`, where we have an outer loop for $\boldsymbol{\lambda}$ and an inner loop that iterates the P-IRLS algorithm till convergence for each trial $\boldsymbol{\lambda}$. For non-Gaussian distribution, the integral in $\mathcal{V}_r$ is not in a closed form. We may approximate the integrand as follows, so that the

Table 4.5: "outer iteration" and "performance iteration" for GAM estimation

"outer iteration"	$\boldsymbol{\lambda}^{[i]}$ ($\boldsymbol{W}^{[k]}$, $\boldsymbol{z}^{[k]}$, $\boldsymbol{\beta}^{[k]}$, $D_p^{[k]}$) $\mathcal{V}_r^{[i]}$, $\boldsymbol{\nabla}\mathcal{V}_r^{[i]}$, $\boldsymbol{\nabla^2}\mathcal{V}_r^{[i]}$
"performance iteration"	$\boldsymbol{W}^{[k]}$, $\boldsymbol{z}^{[k]}$ ($\boldsymbol{\lambda}^{[i]}$, $\boldsymbol{\beta}^{[i]}$, $D_p^{[i]}$, $\mathcal{V}_r^{[i]}$, $\boldsymbol{\nabla}\mathcal{V}_r^{[i]}$, $\boldsymbol{\nabla^2}\mathcal{V}_r^{[i]}$)

integral of the approximated integrand has a closed form.

$$\begin{aligned}
\mathtt{P}(\boldsymbol{y}\,|\boldsymbol{\beta})\mathtt{P}(\boldsymbol{\beta}) &= \exp\{l(\boldsymbol{\beta})\}(2\pi\phi)^{-\frac{p-m}{2}}|\boldsymbol{S}_{\boldsymbol{\lambda}}|_+^{\frac{1}{2}}\exp\{-\tfrac{\boldsymbol{\beta}'\boldsymbol{S}_{\boldsymbol{\lambda}}\boldsymbol{\beta}}{2\phi}\} \\
&= (2\pi\phi)^{-\frac{p-m}{2}}|\boldsymbol{S}_{\boldsymbol{\lambda}}|_+^{\frac{1}{2}}\exp\{l_p(\boldsymbol{\beta},\ \boldsymbol{\lambda})\} \\
&= (2\pi\phi)^{-\frac{p-m}{2}}|\boldsymbol{S}_{\boldsymbol{\lambda}}|_+^{\frac{1}{2}}\exp\{l_s(\phi) - \tfrac{D_p(\boldsymbol{\beta},\ \boldsymbol{\lambda})}{2\phi}\} \\
&\approx (2\pi\phi)^{-\frac{p-m}{2}}|\boldsymbol{S}_{\boldsymbol{\lambda}}|_+^{\frac{1}{2}}\exp\{l_s(\phi) - \tfrac{D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda})+(\boldsymbol{\beta}-\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}})'\boldsymbol{H}_{\boldsymbol{\lambda}}(\boldsymbol{\beta}-\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}})}{2\phi}\}.
\end{aligned}$$

From line 2 to line 3, we substitute $l_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda})$ for $D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda})$ using $D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda}) = 2\big(l_s(\phi) - l_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda})\big)\phi$, where $l_s(\phi)$ is the *saturated log-likelihood*. From line 3 to line 4, we apply a Taylor expansion of $D_p(\boldsymbol{\beta},\ \boldsymbol{\lambda})$ at its minimum point $\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}$. Integrating the approximated integrand gives a Laplace approximation to the integral:

$$\int \mathtt{P}(\boldsymbol{y}\,|\boldsymbol{\beta})\mathtt{P}(\boldsymbol{\beta})\,\mathrm{d}\boldsymbol{\beta} = |\boldsymbol{S}_{\boldsymbol{\lambda}}|_+^{\frac{1}{2}}\exp\{l_s(\phi) - \tfrac{D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda})}{2\phi}\}(2\pi\phi)^{\frac{m}{2}}|\boldsymbol{H}_{\boldsymbol{\lambda}}|^{-\frac{1}{2}},$$

hence the REML score is

$$\mathcal{V}_r(\boldsymbol{\lambda},\ \phi) = -m\log(2\pi\phi) - 2l_s(\phi) + \tfrac{D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda})}{\phi} + \log|\boldsymbol{H}_{\boldsymbol{\lambda}}| - \log|\boldsymbol{S}_{\boldsymbol{\lambda}}|_+.$$

This expression looks like the REML score (1.9) for additive models, but is in fact substantially different. Notably, at convergence of P-IRLS, there is $\boldsymbol{H}_{\boldsymbol{\lambda}} = \boldsymbol{X}'\boldsymbol{W}_{\boldsymbol{\lambda}}\boldsymbol{X} + \boldsymbol{S}_{\boldsymbol{\lambda}}$ where the weight matrix has dependency on $\boldsymbol{\lambda}$. Furthermore, the dependency of $D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda})$ on $\boldsymbol{\lambda}$ is not just via its dependency on $\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}$, but also via its dependency on $\boldsymbol{z}_{\boldsymbol{\lambda}}$ and $\boldsymbol{W}_{\boldsymbol{\lambda}}$. As a result, derivatives of $D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda})$ and $\log|\boldsymbol{H}_{\boldsymbol{\lambda}}|$ w.r.t. $\boldsymbol{\rho} = \log(\boldsymbol{\lambda})$ would be computed in a different way than what is presented in §4.3.5 and §4.3.6 for additive models. See Wood (2011) for full details.

Table 4.5 shows an intuitive comparison between the structures of "outer iteration" and "performance iteration". Covergence of "outer iteration" is guaranteed as a minimum is well defined for both of its outer loop and inner loop. However, since the inner loop needs to update the weighted least squares problem, its computational complexity is $O(np^2) + O(p^3)$. For large n (and possibly big p as well), "outer" iteration is expensive or even infeasible. "Performance iteration" is in principle less costly than "outer iteration", as its inner loop only has a $O(p^3)$ complexity once the QR reduction is done in the outer loop. But it does not always converge, since the REML score is associated with a working additive model that changes at each iteration of the outer loop. It is sometimes possible for the algorithm to cycle around a small set of smoothing parameter, coefficient combinations without ever converging. In this situation, it is helpful to reduce k values of splines to reduce the flexibility of the model. Still, "performance iteration" is useful for GAM estimation with large datasets.

4.5 Summary

Note that it should be emphasized that that $\boldsymbol{W}$ is loop-invariant is the premise for applying model matrix reduction. This is generally true for non-weighted additive models, or models with known weights. If for example, the model error is an AR(1) process, the autocorrelation coefficient has to be known. In other words, if this autocorrelation coefficient needs be estimated, it should not be estimated jointly with smoothing parameters in the REML estimation loop, otherwise the condition for applying model matrix reduction is invalidated.

Chapter 5

Optimizing GAM computations for high performance computing

In the last Chapter I reviewed the computational engine for estimating an additive model. The aim of this Chapter is to investigate where it can be potentially optimized, so that the computational hurdle encountered in Table 3.11 for fitting logBS models can be overcome. To start with, I break down the model fitting time in that table to reveal how much time is spent for pseudo QR reduction and each step of REML estimation. See Table 5.1. The rest of this Chapter is about how to, step by step, obtain better figures in Table 5.2. Here is a brief outline of these sections.

- §5.1 points out a few inefficiencies in current `bam`'s implementation for REML estimation. As mentioned at the beginning of §4, I realized a few places where the design idea could be better implemented when reviewing the computational engine. So in this section I will benchmark my improved version with the original version. As you can see from Table 5.2, the execution time for $\boxed{\log|\boldsymbol{S}_{\boldsymbol{\lambda}}|_+}$ and $\boxed{D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda})}$ steps are successfully reduced.
- §5.2 introduces a scientific library called *Basic Linear Algebra Subroutines (BLAS)* and its optimized distributions, explains concepts of block algorithms and data caching that are related to high performance computing, then exploits such library for GAM computations. As you can see from Table 5.2, except for the $\boxed{\boldsymbol{V}_{\boldsymbol{\lambda}}}$ step, this yields big speedup.
- §5.3 identifies why BLAS has failed to boost $\boxed{\boldsymbol{V}_{\boldsymbol{\lambda}}}$ step. It turns out that the underlying `C` code in `bam` for $\boldsymbol{V}_{\boldsymbol{\lambda}}$ computation is not using BLAS as building blocks so changing BLAS has no performance impact on it. I then re-implement this step using an existing LAPACK routine, and as you can see from Table 5.2, this delivers another big speedup.
- §5.4 explains how $\boxed{\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}}$ step, the slowest part after previous optimizations, can be accelerated. The key is to solve penalized least squares with pivoted Cholesky factorization instead of pivoted QR factorization, which not only involves less FLOP count, but also better benefits from the performance gains from using an optimized BLAS.
- §5.5 makes a final push on computational speed by using parallel computing. Different parallel computing strategies are used for pseudo QR reduction and REML estimation. For the former, the "fork-join" method previously explained in §4.2.3 is used, and for the latter, multi-threading option in an optimized BLAS is enabled. Table 5.2 demonstrates the computation time for model 3.6 when 8 CPU cores are used. Model fitting time is now down to just about 6 minutes.

In §5.5, it will also be demonstrated that computational methods developed in this Chapter will be sufficiently fast to estimate a joint model for logBS from all days of week in 1967. To demonstrate the computational superiority of such new GAM fitting methods, a quick comparison with INLA is made in the summary section. It turns out that fitting daily logBS models via INLA is completely infeasible.

Table 5.1: A break-down of model fitting time (in seconds) previously reported in Table 3.11 for model 3.3b, model 3.5 and model 3.6. Two remarks: 1) Pseudo QR reduction is used as model matrix reduction method, as I have concluded in §4.2.4 that this method outperforms QR reduction; 2) The $\hat{\boldsymbol{y}}$ column refers to the computation of fitted values $\hat{\boldsymbol{y}} = \boldsymbol{X}\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}$ and residuals of a model, as well as other simple statistics like estimated residual variance, AIC, etc. These models are fitted on an Intel Xeon E5-2650 v2 workstation (see Appendix A for hardware information), with `R` software linked to the reference BLAS. BLAS will be shortly introduced in §5.2. The early reference to it here is just to provide full information on how those timing statistics were collected.

	Manchester 11	annual mean logBS	Monday logBS in 1967
	model 3.3b $n = 15707$ $p = 732$	model 3.5 $n = 24239$ $p = 6750$	model 3.6 $n = 54386$ $p = 10932$
pseudo QR reduction	96.66	704.91	4046.42
$\boxed{\log \lvert \boldsymbol{S}_{\boldsymbol{\lambda}} \rvert_+}$	95.76	2518.72	7732.10
$\boxed{\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}}$	174.72	4232.64	23509.83
$\boxed{\boldsymbol{V}_{\boldsymbol{\lambda}}}$	28.56	842.16	4356.09
$\boxed{D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda})}$	16.80	70.24	186.91
$\boxed{\log \lvert \boldsymbol{H}_{\boldsymbol{\lambda}} \rvert}$	31.92	1523.20	6043.37
$\hat{\boldsymbol{y}}$	0.73	35.81	87.88
total	442.15	9927.68	45962.61

Table 5.2: A summary of how modelling estimation is accelerated by using method described in each section of this Chapter. Numbers in the table are execution time measured in seconds. A blank field in the table means that the execution time is not speeded up (nor slowed down of course) by a method, so it is as same as the last non-empty value in a column. All models are fitted on an Intel Xeon E5-2650 v2 workstation (see Appendix A for hardware information). Starting from §5.2, OpenBLAS is linked to `R`.

(a) Results for model 3.3b for logBS time series from *Manchester 11*. This model has an AR(1) component to estimate, so there is an outer iteration via golden-section search.

	golden-section search converges in 14 steps							
	REML estimation converges in 12 Newton steps							
	reduction	$\boxed{\log \lvert \boldsymbol{S}_{\boldsymbol{\lambda}} \rvert_+}$	$\boxed{\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}}$	$\boxed{\boldsymbol{V}_{\boldsymbol{\lambda}}}$	$\boxed{D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda})}$	$\boxed{\log \lvert \boldsymbol{H}_{\boldsymbol{\lambda}} \rvert}$	$\hat{\boldsymbol{y}}$	total
initial	93.66	95.76	174.72	28.56	16.80	31.92	0.73	442.15
§5.1		16.80			0.60			346.99
§5.2	14.56	2.52	45.96		0.44	4.37	0.37	96.78
§5.3				4.70				72.92
§5.4			4.87					31.83

(b) Results for model 3.5 for annual mean logBS. This model has no AR(1) component to estimate.

	REML estimation converges in 8 Newton steps							
	reduction	$\boxed{\log \lvert \boldsymbol{S}_{\boldsymbol{\lambda}} \rvert_+}$	$\boxed{\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}}$	$\boxed{\boldsymbol{V}_{\boldsymbol{\lambda}}}$	$\boxed{D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda})}$	$\boxed{\log \lvert \boldsymbol{H}_{\boldsymbol{\lambda}} \rvert}$	$\hat{\boldsymbol{y}}$	total
initial	704.91	2518.72	4232.64	842.16	70.24	1523.20	35.81	9927.68
§5.1		1066.80			1.44			8406.96
§5.2	85.15	116.80	1540.08		0.96	164.32	25.12	2774.59
§5.3				99.12				2031.55
§5.4			70.56					562.03

(c) Results for model 3.6 for Monday logBS in year 1967. This model has no AR(1) component to estimate.

	REML estimation converges in 10 Newton steps							
	reduction	$\boxed{\log \lvert \boldsymbol{S}_{\boldsymbol{\lambda}} \rvert_+}$	$\boxed{\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}}$	$\boxed{\boldsymbol{V}_{\boldsymbol{\lambda}}}$	$\boxed{D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda})}$	$\boxed{\log \lvert \boldsymbol{H}_{\boldsymbol{\lambda}} \rvert}$	$\hat{\boldsymbol{y}}$	total
initial	4046.43	7732.10	23509.83	4356.09	186.91	6043.37	87.88	45962.61
§5.1		2470.95			4.58			40519.13
§5.2	388.07	246.66	8691.42		3.10	662.88	54.49	13902.71
§5.3				504.00				10550.62
§5.4			281.20					2140.40
§5.5	69.30	53.57	61.05	69.41	1.63	108.62	9.73	373.30

5.1 A better implementation of the existing computational engine

Inefficiencies in the original computation engine are mainly associated with $\boxed{\log|\boldsymbol{S}_{\boldsymbol{\lambda}}|_+}$ and $\boxed{D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda})}$ steps in REML estimation. They have immediate negative impact on speed as they involve redundant computations. $\boxed{\log|\boldsymbol{H}_{\boldsymbol{\lambda}}|}$ step can be optimized to reduce memory footprint. Its improvement won't be practically observed from execution time, as long as RAM is adequate on the testing platform (which is the case on the Intel Xeon E5-2650 v2 workstation I am working with). See Table 5.2 for the practical effects of these improvments.

5.1.1 How $\boxed{\log|\boldsymbol{S}_{\boldsymbol{\lambda}}|_+}$ can be improved

Let us first consider the $\boxed{\log|\boldsymbol{S}_{\boldsymbol{\lambda}}|_+}$ step. As discussed in §4.3.7, the Frobenius norm $\lambda_i\|\boldsymbol{S}_{\boldsymbol{i}}\|_F$ of each individual penalty matrix is computed, and whenever a "bad scaling" between penalty matrices is detected under some numerical tolerance, a similarity transformation is in place. For some reason, implementation in `mgcv` has an interface (see function `gam.reparam`) that only takes the "root" $\boldsymbol{S}_{\boldsymbol{i}}^{\frac{1}{2}}$ (see §4.1.5 if you forget what a "root" is) rather than both $\boldsymbol{S}_{\boldsymbol{i}}$ and $\boldsymbol{S}_{\boldsymbol{i}}^{\frac{1}{2}}$ as input. In consequence, $\boldsymbol{S}_{\boldsymbol{i}}$ is re-computed from $\boldsymbol{S}_{\boldsymbol{i}}^{\frac{1}{2}}$ at the beginning which could have been avoided. There are also other occasions where using $\boldsymbol{S}_{\boldsymbol{i}}^{\frac{1}{2}}$ leads to inefficiency.

1. When similarity transformation $\dot{\boldsymbol{U}}$ is required, $\boldsymbol{S}_{\boldsymbol{i}}$ needs be transformed, too (see §4.3.7). In the general algorithm for $q > 2$ penalty matrices, $\dot{\boldsymbol{U}}$ is updated iteratively. `mgcv` chooses to update $\boldsymbol{S}_{\boldsymbol{i}}^{\frac{1}{2}}$ along this process, but a more efficient way is to apply a one-off transformation $\dot{\boldsymbol{S}}_{\boldsymbol{i}}^{\frac{1}{2}} = \dot{\boldsymbol{S}}_{\boldsymbol{i}}^{\frac{1}{2}}\dot{\boldsymbol{U}}$ in the end, then compute $\dot{\boldsymbol{S}}_{\boldsymbol{i}} = \dot{\boldsymbol{S}}_{\boldsymbol{i}}^{\frac{1}{2}\prime}\dot{\boldsymbol{S}}_{\boldsymbol{i}}^{\frac{1}{2}}$.
2. When computing second derivatives, `mgcv` computes $\dot{\boldsymbol{S}}^{-1}\dot{\boldsymbol{S}}_{\boldsymbol{i}}{}^{\frac{1}{2}\prime}\dot{\boldsymbol{S}}_{\boldsymbol{i}}{}^{\frac{1}{2}}$ by two matrix multiplications instead of just one multiplication $\dot{\boldsymbol{S}}^{-1}\dot{\boldsymbol{S}}_{\boldsymbol{i}}$. In particular, `mgcv` ignores the triangular structure of $\dot{\boldsymbol{S}}_{\boldsymbol{i}}{}^{\frac{1}{2}}$ (after column permutation), and performs the multiplication as if it is a full dense square matrix. As a result, computations of derivatives are twice as much expensive as what is necessary.

There is yet another re-computation. The $\boldsymbol{E}_{\boldsymbol{\lambda}}^*$ (see §4.3.3) required by $\boxed{\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}}$ step can be produced in $\boxed{\log|\boldsymbol{S}_{\boldsymbol{\lambda}}|_+}$ step. But then `mgcv` re-computes $\dot{\boldsymbol{S}}$ by $\boldsymbol{E}_{\boldsymbol{\lambda}}^{*\prime}\boldsymbol{E}_{\boldsymbol{\lambda}}^*$ which is counter-intuitive since it has just obtained $\boldsymbol{E}_{\boldsymbol{\lambda}}^*$ from $\dot{\boldsymbol{S}}$ via a Cholesky factorization!

In summary, the original implementation of $\boxed{\log|\boldsymbol{S}_{\boldsymbol{\lambda}}|_+}$ step is fairly sub-optimal. Even when a similarity transformation turns out unnecessary, it will still perform some meaningless re-computations. I have modified `gam.reparam` so that it takes both $\boldsymbol{S}_{\boldsymbol{i}}$ and $\boldsymbol{S}_{\boldsymbol{i}}^{\frac{1}{2}}$ as input. I also choose the more efficient Cholesky method to compute derivatives, instead of the original QR method (see §4.3.2 for details of both methods if you need a revision).

5.1.2 How $\boxed{D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda})}$ can be improved

In `bam`, the five computation steps for REML estimation are actually arranged in a different order (from what I demonstrated in §4.3) as $\boxed{\log|\boldsymbol{S}_{\boldsymbol{\lambda}}|_+}$, $\boxed{\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}}$, $\boxed{D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda})}$, $\boxed{\boldsymbol{V}_{\boldsymbol{\lambda}}}$ and $\boxed{\log|\boldsymbol{H}_{\boldsymbol{\lambda}}|}$, where $\boxed{D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda})}$ step comes before $\boxed{\boldsymbol{V}_{\boldsymbol{\lambda}}}$ step. In this way, $\boldsymbol{V}_{\boldsymbol{\lambda}}$ is not available for computing the Jacobian matrix $\boldsymbol{J}$ (see §4.3.5). Instead, `bam` computes $\boldsymbol{J}$ solving two triangular systems $\boldsymbol{J} = -\boldsymbol{R}_{\boldsymbol{\lambda}}^{*-1}(\boldsymbol{R}_{\boldsymbol{\lambda}}^{*\prime-1}\boldsymbol{A})$, where $\boldsymbol{R}_{\boldsymbol{\lambda}}^*$ is the 'R' factor from the QR factorization in $\boxed{\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}}$ step (see §4.3.3). Due to the column permutation associated with $\boldsymbol{R}_{\boldsymbol{\lambda}}^*$, rows of $\boldsymbol{A}$ will be reordered when solving these systems, thus the separation between dense blocks and zero blocks in $\boldsymbol{A}$ (see Figure 4.10) are destroyed so that the sparsity of $\boldsymbol{A}$

can not be exploited. As a result, `bam` treats $\boldsymbol{A}$ as a dense matrix, making computations much more expensive.

The "fix" is just as what is described in §4.3). Given that $\boldsymbol{V_\lambda}$ is required anyway by $\boxed{\log|\boldsymbol{H_\lambda}|}$ step, it is more beneficial to schedule its computation ahead to facilitate computations in $\boxed{D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda})}$ step.

Another inefficiency in $\boxed{D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda})}$ step relates to the computation of the Hessian matrix $\frac{\partial^2 D_p}{\partial\boldsymbol{\rho}\partial\boldsymbol{\rho}'}$. `bam` computes this matrix element-wise using a `for`-loop based on formula (4.7). However, as has been demonstrated in that section, writing the result in a matrix form reveals that many terms will cancel out, and the Hessian matrix can be more efficiently obtained using formula (4.8), which requires no loop at all.

5.1.3 How $\boxed{\log|\boldsymbol{H_\lambda}|}$ can be improved

This is comparatively a minor issue, but can help reduce memory usage during REML estimation. Consider the computation of $\mathtt{tr}(\boldsymbol{C_i}\boldsymbol{C_j})$ in $\boxed{\log|\boldsymbol{H_\lambda}|}$ step. `bam` performs the computation at `R`-level rather than at `C`-level. In `R`, arithmetic operations can not be performed on submatrices directly. (This relates to how "vectorization" is performed in `R`.)

- For example, the `R` code `sum(A[1:10, 1:10])` is actually doing `tmp <- A[1:10, 1:10]` and `sum(tmp)`. That is, the submatrix has to be first extracted to a temporary matrix.
- As another example, the `R` code `A[1:10, 1:10] <- A[1:10, 1:10] + 1` is actually doing `tmp <- A[1:10, 1:10] + 1` and `A[1:10, 1:10] <- tmp`. That is, the replacement submatrix has to be first created as a temporary matrix then be copied into the full matrix.

In Figure 4.11 it has been derived that $\mathtt{tr}(\boldsymbol{C_i}\boldsymbol{C_j}) = \mathtt{tr}(\bar{\boldsymbol{C}}_{\boldsymbol{i}}\bar{\boldsymbol{C}}_{\boldsymbol{j}}) = \sum_s\sum_t \bar{\boldsymbol{C}}_{\boldsymbol{i}}(s,\ t)\bar{\boldsymbol{C}}'_{\boldsymbol{j}}(s,\ t)$, where $\bar{\boldsymbol{C}}_{\boldsymbol{i}}$ and $\bar{\boldsymbol{C}}_{\boldsymbol{j}}$ are submatrices of $\boldsymbol{C_i}$ and $\boldsymbol{C_j}$. If this computation is done at `R`-level, there will be too many temporary matrices created (as there is a double loop nest over i and j). In addition, `R` has to do that matrix transpose explicitly. A better idea is to write compiled code like `C` code to directly programme this trace computation. For example, if $\bar{\boldsymbol{C}}_{\boldsymbol{i}}$ is in row u_1 to u_2 of $\boldsymbol{C_i}$ and $\bar{\boldsymbol{C}}_{\boldsymbol{j}}$ is in row v_1 to v_2 of $\boldsymbol{C_j}$, the trace is just $\sum_{s=u_1}^{u_2}\sum_{t=v_1}^{v_2}\boldsymbol{C_i}(s,\ t)\boldsymbol{C_j}(t,\ s)$, eliminating the need to extract $\bar{\boldsymbol{C}}_{\boldsymbol{i}}$ and $\bar{\boldsymbol{C}}_{\boldsymbol{j}}$.

5.2 Using optimized BLAS for high performance computing

5.2.1 An introduction to BLAS

Almost certainly, if you want to do high performance computing, you can't miss the topic of BLAS.

BLAS (Basic Linear Algebra Subprograms) is a set of standard specifications on a group of elementary numerical operations for common linear algebra computations. It includes the level-1 BLAS (Lawson et al., 1979) for scalar-vector and vector-vector operations, the level-2 BLAS (Dongarra et al., 1988) for matrix-vector operations and the level-3 BLAS (Dongarra et al., 1990) for matrix-matrix operations. These specifications were optimally designed, providing sufficiently flexible functionality that meets practical needs for programming linear algebra computations, while keeping the set of elementary operations as small as possible. For example, the operation `daxpy` from level-1 BLAS does $\boldsymbol{y} = \boldsymbol{y}+\alpha\boldsymbol{x}$, that is, multiplying a scalar α to a vector x then adding the resulting vector to $\boldsymbol{y}$. The operation `dgemv` from level-2 BLAS does $\boldsymbol{y} = \alpha\mathtt{op}(\boldsymbol{A})\boldsymbol{x}+\beta\boldsymbol{y}$, where $\mathtt{op}(\boldsymbol{A}) = \boldsymbol{A}'$ or $\boldsymbol{A}$ is a matrix, $\boldsymbol{x}$ and $\boldsymbol{y}$ are vectors,

and α and β are scalars. The operation `dgemm` from level-3 BLAS does $\boldsymbol{C} = \alpha\texttt{op}(\boldsymbol{A})\texttt{op}(\boldsymbol{B}) + \beta\boldsymbol{C}$, where $\boldsymbol{A}$, $\boldsymbol{B}$ and $\boldsymbol{C}$ are matrices, and α and β are scalars. Together with such specifications, a model implementation written in FORTRAN 77 is provided by Netlib (a repository of software for scientific computing, maintained by AT&T, Bell Laboratories, the University of Tennessee and Oak Ridge National Laboratory). These model subprograms / subroutines / functions further introduce parameters like "stride" and "leading dimension", to enable operations on vectors whose elements are not necessarily stored contiguously and submatries from a larger matrix. Such model implementation serves as a stand-alone library, commonly known as the "F77 BLAS" or the "reference BLAS", and it is portable on all machines.

However, the motivation of BLAS standard, is that BLAS library should be optimized for high performance on any particular machine, by making optimal use of its hardware. The reference BLAS should only serves a sanity check, once an optimized implementation is available. Optimization are often provided by machine's vendor (but maybe as a proprietary library), like the MKL (Math Kernel Library) (Burylov et al., 2007) from Intel. But a number of open source optimized BLAS libraries are also available and offering competitive performance. These include ATLAS (Automatically Tuned Linear Algebra Software) (Whaley et al., 2000) and OpenBLAS (Zhang et al., 2011) (a project aiming to extend and maintain the successful GotoBLAS (Goto and Geijn, 2008a,b)).

Optimization of BLAS is important and thriving, because BLAS operations, as they were supposed to be, are building blocks for more complicated operations that forms advanced numerical linear algebra libraries. A popular example is the LAPACK (Linear Algebra Package) (Demmel et al., 1987; Anderson et al., 1990, 1999; Anderson and Dongarra, 1990), aiming at solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, singular value problems, as well as their associated matrix factorizations like LU, Cholesky, QR, SVD, Schur and generalized Schur. With a high-performance BLAS library, LAPACK operations automatically gains efficiency.

All matrix computations introduced in §4.1 can be implemented with existing level-3 BLAS subroutines or LAPACK routines. For demonstration, let $\boldsymbol{A}$ and $\boldsymbol{B}$ be square matrices, $\boldsymbol{S}$ be a positive-definite matrix, $\boldsymbol{R}$ be a full-rank upper triangular matrix, $\boldsymbol{P}$ be a column permutation matrix and $\boldsymbol{Q}$ be the 'Q' factor (in factored form storage) from a QR factorization, then

- the matrix cross-product $\boldsymbol{A'A}$ (or $\boldsymbol{AA'}$) can be computed using level-3 BLAS subroutine `dsyrk`;
- the matrix multiplication $\boldsymbol{AB}$ (or $\boldsymbol{A'B}$, $\boldsymbol{AB'}$, $\boldsymbol{A'B'}$) can be computed using level-3 BLAS subroutine `dgemm`;
- triangular matrix multiplication $\boldsymbol{RB}$ (or $\boldsymbol{R'B}$) can be computed using level-3 BLAS subroutine `dtrmm`;
- solving a triangular system of equations $\boldsymbol{R}^{-1}\boldsymbol{B}$ (or $\boldsymbol{R'}^{-1}\boldsymbol{B}$) can be computed using level-3 BLAS subroutine `dtrsm`;
- pivoted QR factorization $\boldsymbol{AP} = \boldsymbol{QR}$, can be computed using LAPACK routine `dgeqp3`;
- orthonormal transformation $\boldsymbol{Q'B}$ can be computed using LAPACK routine `dormqr`;
- pivoted Cholesky factorization $\boldsymbol{P'SP} = \boldsymbol{R'R}$ can be computed using LAPACK routine `dpstrf`;
- positive-definite matrix inverse $\boldsymbol{S}^{-1}$ can be computed by LAPACK routine `dpotri`, which consists of a call to `dtrtri` for triangular matrix inverse and a call to `dlauum` for transposed triangular matrix cross-product;
- symmetric eigen decomposition can be computed using LAPACK routine `dsyevr`.

As with many scientific software, `R` relies on BLAS and LAPACK to do linear algebra computations. For example, the `R` function `"%*%"` for matrix multiplication is mapped to the level-3 BLAS subroutine

`dgemm`. Particularly, BLAS library is a shared library that is dynamically linked to `R` and automatically loaded at `R`'s startup time. Such dynamic linking can be altered at any time by a user, so it is straightforward to experiment `R` programs with different BLAS libraries to compare performance. Very often, `R` is linked to the reference BLAS library, because many users (including me) don't have an exisiting BLAS library on their machines prior to `R` installation, thus `R` will create a reference BLAS library for its use. Therefore, linking `R` to an optimized BLAS library, like MKL and OpenBLAS, are very likely to yield performance boost to a program, if it is rich in BLAS and LAPACK operations.

5.2.2 Benchmarking BLAS

After installing an optimized BLAS library, the first thing to do is benchmarking. In this section, I will compare the performance of the matrix computations listed above in `R`, using reference BLAS, OpenBLAS and MKL (if available). Hardware information on tested machines is given in Appendix A. Here, performance is reported in GFLOPs (short for GigaFLOPs). 1 GFLOPs = 10^9 FLOPs, and FLOPs means "floating-point operations per second". Apart from symmetric eigen decomposition, the exact FLOP count for all matrix computations in the list is explicitly known given dimension of the matrices. The performance can then be derived by dividing FLOP count by measured computation time.

Note that an optimized BLAS library is often multi-threaded to exploit multi-core hardware for parallel computing, but there are several different ways to control the number of threads accessible to the library. To start with, I will pin the number of threads at 1 for strict serial computing.

Figure 5.1 illustrates the result of my simple benchmarking (symmetric eigen decomposition is excluded as its exact FLOP count is unknown). From the figure, the following observations can be made.

- On all machines, optimized BLAS is more than 10 times faster than the reference version for level-3 BLAS operations. On the Intel Xeon E5-2650 v2 workstation where both MKL and OpenBLAS are available, these two libraries seem to deliver similar performance.
- Performance of LAPACK routines varies between routines. Positive-definite matrix inverse has competitive performance with level-3 BLAS subroutines. Performance of pivoted Cholesky factorization and orthonormal transformation is lower but still good. But pivoted QR factorization has the poorest performance.
- Performance of LAPACK routines also depends on the version of BLAS. On the Intel Xeon E5-2650 v2, MKL outperforms OpenBLAS for orthonormal transformation, but the other way round for pivoted Cholesky factorization.

In the next two sections, I will explain why optimized BLAS is much faster than the reference BLAS, and why LAPACK routines differ in performance.

5.2.3 Block algorithms and data caching

The performance advantage of an optimized BLAS to the reference BLAS comes from the block algorithms it uses for matrix computations to achieve data caching. In scientific computing, using block algorithms is a primary step for high-performance computing as this gives the best cache performance (Lam et al., 1991).

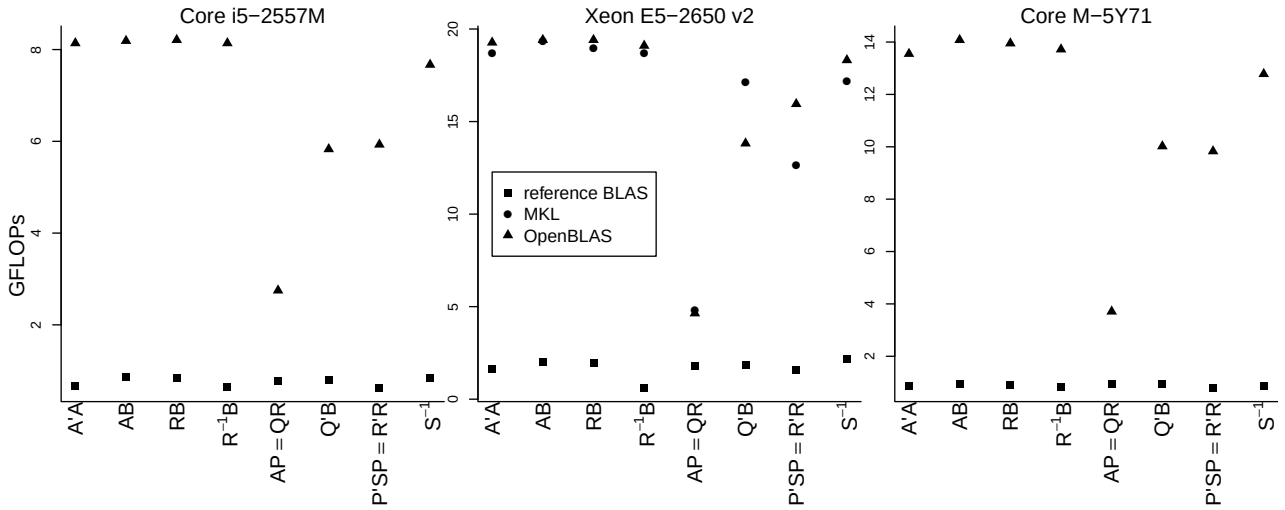


Figure 5.1: BLAS and LAPACK's single-threading performance (in GFLOPs)

Our computers have many data storage layers. The closest layer to CPU is a set of CPU registers. CPU can access data on these units with no delay (a more appropriate term is "latency"). Registers are where CPU performs computations. All data must be bring to registers before CPU can process them. However, registers are very small in numbers and size. Even holding a 4×4 matrix is impossible. So data must be frequently exchanged between registers and other memory layers during computations: those requested by the current instruction are read in, and those not to be used within the next few instructions will be kicked out. A CPU cache, or L1 (level-1) data cache, is a secondary memory layer right under the registers. It has a bigger size (normally 32KB) and is also fast enough. You can put in a double-precision floating-point vector of 4096 elements, or put in a 64×64 matrix, or a few smaller matrices (say three 36×36 matrices). Below this CPU cache there may be other cache layers, like L2 cache; then below all cache layers is the main RAM. During computation, data will be frequently transported between all layers. When CPU requests a datum, the farther it is from registers, the longer time it takes to fetch it to registers. Furthermore, different layers have difference access speed (or latency). It takes at most 1 CPU cycle to access a register, then normally 4 cycles to access L1 data cache, then maybe 20 cycles to access L2 data cache, and finally, 80 to 200 cycles to access main RAM. In consequence, how fast a program can run depends on how fast data CPU can obtain data it demands. Algorithms with different data access pattern generally give different performance.

For example, to compute a matrix multiplication $\boldsymbol{C} = \boldsymbol{AB}$ between two 1200×1200 square matrices $\boldsymbol{A}$ and $\boldsymbol{B}$, consider the following two arrangements.

1. Compute 1200 matrix-vector multiplications $\boldsymbol{C}(,\ j) = \boldsymbol{AB}(,\ j)$, updating $\boldsymbol{C}$ column by column;
2. Partition each matrices into 40 small 30×30 blocks like

$$\boldsymbol{A} = \begin{pmatrix} \boldsymbol{A_{1,1}} & \boldsymbol{A_{1,2}} & \cdots & \boldsymbol{A_{1,40}} \\ \boldsymbol{A_{1,1}} & \boldsymbol{A_{1,2}} & \cdots & \boldsymbol{A_{1,40}} \\ \vdots & \vdots & \vdots & \\ \boldsymbol{A_{40,1}} & \boldsymbol{A_{40,2}} & \cdots & \boldsymbol{A_{40,40}} \end{pmatrix}$$

   and do block matrix-matrix multiplication $\boldsymbol{C_{I,J}} = \sum_{K=1}^{40} \boldsymbol{A_{I,K}} \boldsymbol{B_{K,J}}$ to update $\boldsymbol{C}$ block-by-block.

In the first case, each iteration over j-loop needs to access the whole matrix $\boldsymbol{A}$. A 1200×1200 matrix is far too large to be retained in either registers or L1 data cache. After $\boldsymbol{C}(,\ j)$ is updated, most elements of $\boldsymbol{A}$ would have been evicted from L1 cache, so that $\boldsymbol{A}$ needs be loaded from lower memory layers for the update of $\boldsymbol{C}(,\ j+1)$. The second case appears to be less efficient as it involves more

levels of loop, however, L1 cache is large enough to hold $\boldsymbol{A_{I,K}}$, $\boldsymbol{B_{K,J}}$ and $\boldsymbol{C_{I,J}}$. There will be no data eviction from L1 cache to lower memory layers during computation of $\boldsymbol{C_{I,J}}$. If you make a careful count, you will find that in the first case, each element of $\boldsymbol{A}$ is read from memory layers below L1 cache 1200 times, while in the second case, this value is only 40. In the second case, CPU can get data it needs faster, so the matrix-matrix multiplication is faster.

The second algorithm is just an example of a block algorithm. It has an effect of "data caching" (each element of $\boldsymbol{A}$ is reused 30 times in L1 cache before being kicked out) and has a higher performance. See Drepper (2007, §3) and Hennessy and Patterson (2011, Chapter 2) for more about CPU cache, and Kowarschik and Weiß (2003) for an overview of block algorithms and data caching in numerical computations.

The above simplified matrix-matrix multiplication example just explains why an optimized BLAS is faster than the reference BLAS. An optimized BLAS implements block algorithms for level-3 BLAS operations. They have a `dgemm`-based structure, by highly optimizing matrix-matrix multiplication and rebuilding other level-3 BLAS operations using the optimized `dgemm`. Such design idea dates back to Kågström et al. (1998), then was superseded by van de Geijn and Quintana-Ortí (2008, Chapter 5) and Gunnels et al. (2001), and was recently reinterpreted by Van Zee and van de Geijn (2015) and Van Zee et al. (2016).

In a block algorithm, there is one or more parameters called *blocking factors*. In the above matrix-matrix multiplication example, 30 is a (and the only) blocking factor. Blocking factors affect the effectiveness of caching. Too large a value destroys caching while too small a value underuses cache capacity.

It is worth pointing out that within BLAS, only level-3 operations can substantially benefit from block algorithms. The purpose of a block algorithm is to achieve data resue and that reuse takes place in L1 cache. Level-1 BLAS for vector-vector operations has no data resue at all as all vectors are only visited once. Level-2 BLAS for matrix-vector operations has no data reuse for the matrix operand. The vector operand may be reused but the gains in performance won't be substantial, because the computational complexity of a matrix-vector operation is determined by the dimension of the matrix. For example, the optimized matrix-vector multiplication from OpenBLAS only has 2.9 GFLOPs performance on an Intel Xeon E5-2650 v2, but the matrix-matrix multiplication attains 19.41 GFLOPs performance. In fact, it is exactly this difference between three levels of operations that makes LAPACK routines differ greatly in performance.

5.2.4 Understanding the performance difference between LAPACK routines

Block algorithms are also used for matrix factorizations, although they are generally more complicated. Such algorithms are constantly developed for LAPACK and *LAPACK Working Notes* has collected hundreds of research work on the theme. LAPACK aims to reorganize basic computational algorithms to use level-3 BLAS operations as much as possible, ending up with block matrix computations in the innermost loop. In this way, the high performance of an optimized BLAS can be carried as much as as possible.

In §4.1 I have explained algorithms for some matrix computations. They can be easily programmed using level-1 and level-2 BLAS operations. However, these basic algorithms are primarily used for educational purposes and are not those behind optimized level-3 BLAS or LAPACK. For some examples of block algorithms, see Quintana-Ortí et al. (1998) for block pivoted QR factorization, Bischof and Loan (1987) for block orthonormal transformation and Lucas (2004) for block pivoted Cholesky factorization. For positive-definite matrix inverse, the basic algorithm for triangular matrix inverse in Figure 4.3a can be extended to a block version straightforward, treating all elements like R_{ii} as block matrices. Rearranging the algorithm in Figure 4.3b for transposed triangular matrix cross-product

to be block-oriented is less straightforward. I am unable to locate an exact reference for the block implementation in LAPACK, so I encourage all keen readers to read the source code of `dlauum`.

Although a LAPACK routine is intended to be rich in level-3 BLAS operations, some routines still consist of a significant proportion of level-2 BLAS operations (primarily the matrix-vector multiplication). It turns out that for block pivoted QR factorization, there are still exactly 50% of its computations being matrix-vector multiplications. Let r_2 and r_3 respectively be the performance of matrix-vector and matrix-matrix multiplications, then pivoted QR factorization should have an overall performance of $2/(\frac{1}{r_2} + \frac{1}{r_3})$. On an Intel Xeon E5-2650 v2 with OpenBLAS, It can be measured that $r_2 \approx 2.9$ and $r_3 \approx 19$, so pivoted QR factorization is expected to have a performance of 5 GFLOPs, which agrees with measured performance in Figure 5.1.

By contrast, pivoted Cholesky factorization is richer in matrix-matrix multiplications. Let b and t respectively be the blocking factor and the dimension of the matrix, then matrix-matrix multiplication accounts for $(1 - \frac{3b}{2t})$ fraction of its computations (Lucas, 2004). LAPACK would choose $b = 64$ (see `ilaenv` of LAPACK), so for example for an 5000×5000 matrix, this proportion is 98.08% which is almost 100%.

5.2.5 Using optimized BLAS for GAM computations

Following §4.2 and §4.3, it is clear that GAM computations are rich in the matrix computations I have just benchmarked. Thus, using an optimized BLAS library is very promising to speed up model fitting. From Figure 5.1 it appears that OpenBLAS generally outperforms MKL on an Intel Xeon E5-2650 v2 workstation (see Appendix A for hardware information), so I will link `R` with OpenBLAS for testing.

Experiment results are presented in Table 5.2, under row "§5.2". Comapred with with previous row "§5.1", the speedup is not uniform for all computation steps, but mostly agrees with what I have explained in previous sections.

- The acceleration of pseudo QR reduction is most dramatic, as its computations is dominated by matrix cross-product for which an optimized BLAS library attains the best performance.
- For REML estimation, the performance improvement on $\boxed{\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}}$ is limited as it is rich in pivoted QR factorization. Otherwise for $\boxed{\log|\boldsymbol{S}_{\boldsymbol{\lambda}}|_+}$ and $\boxed{\log|\boldsymbol{H}_{\boldsymbol{\lambda}}|}$ steps, the gains is substantial.

However, it is surprising to see that $\boxed{\boldsymbol{V}_{\boldsymbol{\lambda}}}$ has no performance improvement at all. Why?

5.3 Computing $\boldsymbol{V}_{\boldsymbol{\lambda}}$ with BLAS

It turns out that in `bam`, computation of $\boldsymbol{V}_{\boldsymbol{\lambda}}$ is programmed with BLAS-free `C` code, thus changing BLAS has no effects on it. As is listed in §5.2.1, there is an existing LAPACK routine `dpotri` for such computation, so I proceed to use this routine. The experiment result in Table 5.2 proves this very worthwhile.

The `C` routines in `bam` are inferior to `dpotri` in the following aspects.

1. Triangular matrix inverse $\tilde{\boldsymbol{R}} = \boldsymbol{R}^{-1}$ is computed by solving triangular systems, as is covered in §4.1.4. But `dpotri` uses the algorithm in Figure 4.3a with a block implementation for caching.

2. Computation of $\tilde{\boldsymbol{R}}\tilde{\boldsymbol{R}}'$ does not exploit the triangular structure of $\tilde{\boldsymbol{R}}$, as is suggested by the algorithm in Figure 4.3b. The resulting matrix multiplication costs p^3 FLOP rather than $\frac{1}{3}p^3$ FLOP.

5.4 Computing $\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}$ via Cholesky factorization

After previous optimization, $\boxed{\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}}$ is now the slowest part hence the bottleneck of GAM computations.

As is described in 4.3.3, $\boxed{\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}}$ step is dominated by a pivoted QR factorization of a $2p \times p$ augmented model matrix. However, it is already observed in §5.2 that such factorization can not attain high performance with an optimized BLAS library. So a good idea is to find an alternative algorithm than using pivoted QR factorization.

In fact, this is possible. From (1.8) we see that $\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}$ solves a system of linear equations $\boldsymbol{H}_{\boldsymbol{\lambda}}\boldsymbol{\beta} = \boldsymbol{X}'\boldsymbol{W}\boldsymbol{y}$ with a positive definite or positive-semidefinite coefficient matrix $\boldsymbol{H}_{\boldsymbol{\lambda}} = \boldsymbol{X}'\boldsymbol{W}\boldsymbol{X} + \boldsymbol{S}_{\boldsymbol{\lambda}}$. Now consider a pivoted Cholesky factorization with pre-conditioning (see §4.1.2 if you need a revision) $\boldsymbol{H}_{\boldsymbol{\lambda}} = (\boldsymbol{R}^*_{\boldsymbol{\lambda}})'\boldsymbol{R}^*_{\boldsymbol{\lambda}}$ where $\boldsymbol{R}^*_{\boldsymbol{\lambda}} = \boldsymbol{R}_{\boldsymbol{\lambda}}\boldsymbol{R}'_{\boldsymbol{\lambda}}\boldsymbol{J}_{\boldsymbol{\lambda}}$, then $\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}$ can be efficiently obtained by solving two triangular systems (see §4.1.3 if you need a revision) $\tilde{\boldsymbol{z}} = \boldsymbol{R}^{*\prime-1}_{\boldsymbol{\lambda}}\boldsymbol{z}$ and $\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}} = \boldsymbol{R}^{-1}_{\boldsymbol{\lambda}}\tilde{\boldsymbol{z}}$. Such computation only involves $(\frac{1}{3}p^3 + 2p^2)$ FLOP, much lower than the $\frac{10}{3}p^3$ FLOP using the QR factorization method in §4.3.3. In addition, that pivoted Cholesky factorization outperforms pivoted QR factorization in an optimized BLAS implies that the practical gains from using this Cholesky method can be more than what the FLOP count shows. The results in Table 5.2 shows that it is 20 times faster for model 3.5 and 30 times faster for 3.6!

Computations of $D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda})$ and $\log|\boldsymbol{H}_{\boldsymbol{\lambda}}|$ are also convenient in this method. There are $D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda}) = \left\|\boldsymbol{W}^{\frac{1}{2}}\boldsymbol{y}\right\|^2 - \|\tilde{\boldsymbol{z}}\|^2$ and $\log|\boldsymbol{H}_{\boldsymbol{\lambda}}| = 2\sum_{i=1}^{p}\big(\log|\boldsymbol{R}_{\boldsymbol{\lambda}}(i,\ i)| + \log\boldsymbol{J}_{\boldsymbol{\lambda}}(i,\ i)\big)$.

In case $\boldsymbol{H}_{\boldsymbol{\lambda}}$ is rank-deficient with rank $r < p$, computations proceed as follows. Compute $\boldsymbol{b} = \boldsymbol{P}'_{\boldsymbol{\lambda}}\boldsymbol{J}^{-1}_{\boldsymbol{\lambda}}\boldsymbol{z}$. Initialize $\tilde{\boldsymbol{z}}$ as a length-p vector of zeros and set $\tilde{\boldsymbol{z}}(1:r) = \boldsymbol{R}_{\boldsymbol{\lambda}}(1:r,\ 1:r)'^{-1}\boldsymbol{b}(1:r)$. Reset $\boldsymbol{b}(1:r) = \boldsymbol{R}_{\boldsymbol{\lambda}}(1:r,\ 1:r)^{-1}\tilde{\boldsymbol{z}}(1:r)$ and $\boldsymbol{b}\big((r+1):p\big) = \boldsymbol{0}$, then compute $\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}} = \boldsymbol{J}^{-1}_{\boldsymbol{\lambda}}\boldsymbol{P}'_{\boldsymbol{\lambda}}\boldsymbol{b}$, $D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda}) = \left\|\boldsymbol{W}^{\frac{1}{2}}\boldsymbol{y}\right\|^2 - \|\tilde{\boldsymbol{z}}(1:r)\|^2$ and $\log|\boldsymbol{H}_{\boldsymbol{\lambda}}| = 2\sum_{i=1}^{r}\log|\boldsymbol{R}_{\boldsymbol{\lambda}}(i,\ i)| + 2\sum_{i=1}^{p}\log\boldsymbol{J}_{\boldsymbol{\lambda}}(i,\ i)$.

Note that the matrix cross-product $\boldsymbol{X}'\boldsymbol{W}\boldsymbol{X}$ and $\boldsymbol{z} = \boldsymbol{X}'\boldsymbol{W}\boldsymbol{y}$ required by this method are readily accumulated in pseudo QR reduction (see Figure 4.6 if you need to remaster the algorithm); In addition, the final Cholesky factorization in the original pseudo QR reduction will no longer be necessary.

5.5 Using parallel computing for GAM computations

Previously I have been using OpenBLAS for serial computing, now I will investigate further speedup via parallel computing.

5.5.1 Parallel performance of BLAS and LAPACK

Let us start by testing the parallel performance of BLAS and LAPACK. As it turns out, the parallel performance for level-3 BLAS operations has almost perfect parallel scaling (see §4.2.3 if you need a

revision on parallel speedup and parallel scaling): using m CPU cores makes the computations almost m times as fast. So more interest is on comparing parallel performance between LAPACK routines.

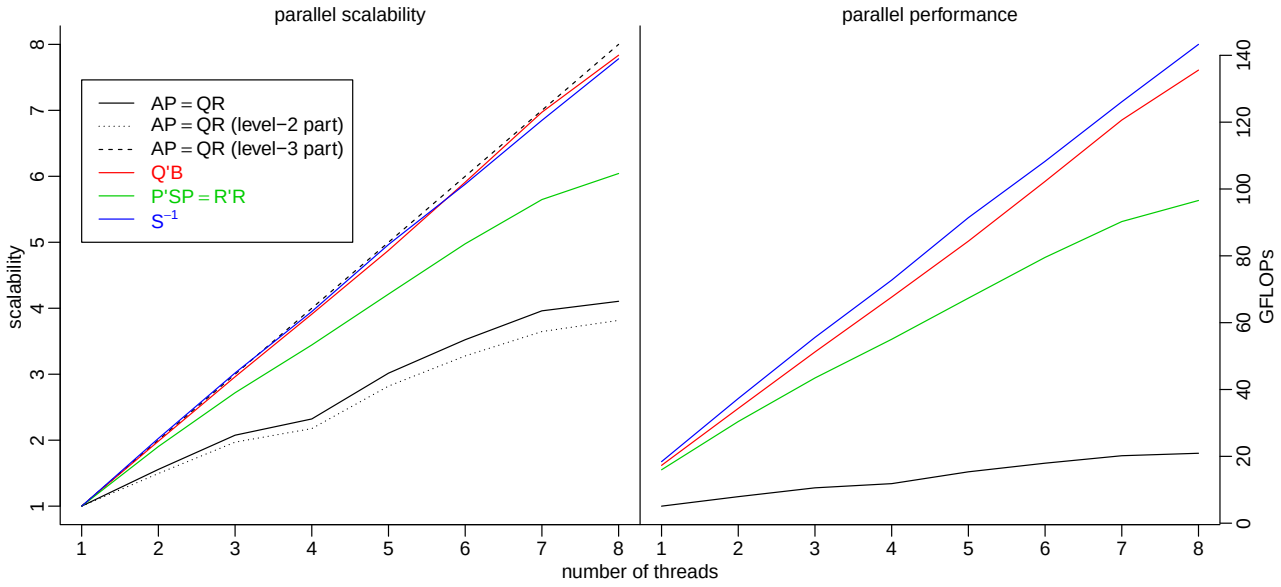


Figure 5.2: Parallel scaling (left panel) and parallel performance (right panel, unit: GFLOPs) for LAPACK operations: pivoted QR factorization (black), orthonormal transformation (red) pivoted Cholesky factorization (green) and positive-definite matrix inverse (blue). Particularly, in the left panel, the scaling for level-2 (black dotted) and level-3 (black dashed) BLAS computations inside pivoted QR are also displayed. Experiment is conducted on an Intel Xeon E5-2650 v2 workstation with OpenBLAS (see Appendix A for hardware information). All matrices used for the experiment are $N_m \times N_m$ square matrix, but the dimension grows with the number of threads m with relationship $N_m = \sqrt[3]{m}N_1$. Since all operations involve $O(N_m^3)$ FLOP, this makes sure the loading for each thread is constant at $O(N_1^3)$ no matter how m grows. For this benchmarking, I have chosen chosen $N_1 = 10000$. The parallel scaling factor measured this way is known as the *weak parallel scaling factor*, since the problem size grows with m. Another type of parallel scaling factor is *strong parallel scaling factor*, where the problem size is held fix. That is, using $N_m = N_1$. Strong scaling is an ideal property, but more difficult to achieve than weak scaling. This is not difficult to understand. If problem size is fixed, then the loading for each thread will diminish as the number of cores increases, while at the same time the overhead for parallel scheduling will increase.

Figure 5.2 illustrates measured parallel performance of the four LAPACK routines listed in §5.2.1. The left panel sketches parallel scaling factor against the number of CPU cores m, and the right panel sketches the measured performance (in GFLOPs) against m. There are the following observations.

- Positive-definimate matrix inverse (blue line) and orthonormal transformation (red line) have near perfect parallel performance;
- The parallel scaling of pivoted Cholesky factorization (green line) is inferior, but still remarkable. There is a 6 times speedup when 8 cores are used;
- The parallel scaling of pivoted QR factorization (black solid line) is poor. Using 8 cores only achieves 4 times speedup, and the scaling curve begins plateaus.

So why does pivoted QR factorization scale so poorly? The reason is still with the large proportion of level-2 BLAS operations it involves. As is mentioned earlier, parallel scaling for level-3 BLAS operations is impressively perfect. However, the scaling for level-2 BLAS operations is not even as half as good. See the left panel of Figure 5.2 again, and look for the black dashed line and the black dotted line. The former is the parallel scaling (good) for the level-3 BLAS computations inside pivoted QR factorization, and the latter is the parallel scaling (poor) for the level-2 BLAS computations. Since level-2 BLAS operations account for 50% of computations of pivoted QR factorization, it limits the parallel scaling of pivoted QR factorization. Other LAPACK operations have demonstrated good parallel scaling, because they involve a smaller proportion of level-2 BLAS operations.

So why does level-2 BLAS scale poorly as opposed to level-3 BLAS? Consider an example of matrix-vector multiplication and matrix-matrix multiplication. For simplicity, all matrices are square of size N. For large N, the former performs $2N^2$ FLOP on $(N^2 + \frac{N}{b} + N) \approx N^2$ data read and N data write, while the latter performs $2N^3$ FLOP on $(2N^2 + \frac{N^3}{b}) \approx \frac{N^3}{b}$ data read and $\frac{N^3}{b}$ data write. Here, b is the blocking factor that MKL or OpenBLAS chooses for caching (see §5.2.3 if you need a revision on blocking factor). The exact derivation of the amount of data motion requires knowledge of the blocking strategy used by optimized BLAS library, but is not our focus here. It is sufficient to observe that the number of FLOP and memory operations is $2:1$ for the former, but $b:1$ for the latter. On an Intel Xeon E5-2650 v2 workstation with OpenBLAS (see Appendix A for hardware information), MKL and OpenBLAS chooses $b \approx 256$ (of course, the exact value is machine dependent), so we can conclude that matrix-vector multiplication is *memory-bound* but matrix-matrix multiplication is *computation-bound* or *CPU bound*. Memory-bound operations inflict heavier front-side bus (FSB) pressure, and the FSB bandwidth can easily become a limiting factor for their parallel scalability, which is particularly true in shared-memory parallel computing via multi-threading. With $N = 10000$, matrix-vector multiplication takes about 0.069 seconds (hence giving 2.9 GFLOP performance as we claimed earlier), implying that data are transported in FSB at a rate of 11.6 GB/s, which is already 77.7% of a memory channel's capacity. In multi-threading, each thread would demand a data flow at this supplying rate. As soon as the gross demand exceeds the maximum FSB bandwidth of all four channels, which is 59.712 GB/s, the FSB is "jam" and CPU cores have to wait longer for data's arrival. Another implication of this, is that the scalability will be upper bounded by $59.712/11.6 \approx 5.15$, however we increase the number of threads, justifying why the black dotted curve in left panel of Figure 5.2 begins to plateau. By contrast, data are only flowing in FSB at 2.53 GB/s for matrix-matrix multiplication. Even if all 16 cores are used, the gross demand is 40.48 GB/s which the FSB has no difficulty to meet. I do make a test with 16 threads, and it turns out that the scaling factor is as high as 15 for matrix-matrix multiplication, but is still held at 4 for matrix-vector multiplication.

A message of this, is that it is a good idea to avoid using pivoted QR factorization for developing algorithms if we wish the algorithm to scale well. Fortunately, the pivoted QR factorization in $\boxed{\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}}$ step of REML estimation has been replaced by pivoted Cholesky factorization in the last section. In addition, I have been using pseudo QR reduction rather than QR reduction, so no pivoted QR factorization is involved in model matrix reduction step either. These give a very positive message that GAM computations should scale reasonbly well in practice. In the next section, I will experiment GAM computations with parallel computing.

5.5.2 Experimenting parallel computing for GAM computations

`bam` already has a "fork-join" implementation for parallel pseudo QR reduction, so for this part, I will supply it with single-threaded OpenBLAS. For REML estimation, I will enable multi-threading for OpenBLAS, so that REML estimation will be "automatically" parallelized.

It is generally a good idea to experiment parallel computing on a problem with big size. I will consider the following **test model**:

$$\begin{aligned}\mathtt{logBS}_{\mathtt{id}} = {} & f_0(\mathtt{E_i};\ 5) + f_1(\mathtt{w_y};\ 20) + f_2(\{\mathtt{e_i},\ \mathtt{n_i}\};\ 200) + f_3(\mathtt{w_y},\ \{\mathtt{e_i},\ \mathtt{n_i}\};\ 20,\ 200) + \\ & f_4(\mathtt{T}^0_{\mathtt{id}};\ 15) + f_5(\mathtt{T}^0_{\mathtt{id}},\ \{\mathtt{e_i},\ \mathtt{n_i}\};\ 15,\ 200) + \\ & f_6(\mathtt{T}^*_{\mathtt{id}};\ 15) + f_7(\mathtt{T}^*_{\mathtt{id}},\ \{\mathtt{e_i},\ \mathtt{n_i}\};\ 15,\ 200) + \\ & f_8(\mathtt{T}^0_{\mathtt{id}},\ \mathtt{T}^*_{\mathtt{id}};\ 15,\ 15) + f_9(\mathtt{i};\ 1276) + f_{10}(\mathtt{w_y};\ 52) + f_{11}(\mathtt{h_i};\ 10) + \\ & f_{12}(\mathtt{d_w};\ 7) + f_{13}(\mathtt{d_w},\ \mathtt{w_y};\ 7,\ 20) + f_{14}(\mathtt{d_w},\ \{\mathtt{e_i},\ \mathtt{n_i}\};\ 7,\ 200) + e_{\mathtt{id}},\end{aligned}$$

This is a very simple estension to model 3.6, as a joint model for logBS from all days of week. The extension procedure is similar to what was previously done in the *Manchester 11* case study. Day of week $\mathtt{d_w}$ is introduced as a factor variable (levels 1 to 7, for Monday to Sunday), and f_1 and f_2 are allowed to vary for each factor level. To be precise, new components to model 3.6 are

Table 5.3: Time (in seconds) for fitting the **test model** assuming i.i.d. model errors (examination of its residuals will be done in the next Chapter). The model has 12460 coefficients for 387253 data. The model is fitted on an Intel Xeon E5-2650 v2 workstation (see Appendix A for hardware information), with `R` software linked to OpenBLAS. Up to 8 CPU cores are used for parallel computing. For pseudo QR reduction, the "fork-join" parallelism is implemented with single-threaded OpenBLAS; for REML estimation, multi-threading is enabled for OpenBLAS. Also see Figure 5.3 for illustration of parallel scaling.

		REML estimation converges after 22 iterations						
m	reduction	$\boxed{\log\lvert \boldsymbol{S}_{\boldsymbol{\lambda}}\rvert_+}$	$\boxed{\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}}$	$\boxed{\boldsymbol{V}_{\boldsymbol{\lambda}}}$	$\boxed{D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda})}$	$\boxed{\log\lvert \boldsymbol{H}_{\boldsymbol{\lambda}}\rvert}$	$\hat{\boldsymbol{y}}$	total
1	3675.00	507.8	850.41	1476.02	7.47	1665.56	714.21	8896.47
2	1914.06	286	471.32	746.37	5.09	882.35	371.39	4676.58
3	1318.00	211.05	344.09	513.57	4.35	619.99	249.97	3261.02
4	1011.64	174.06	280.11	397.29	4.02	489.75	192.84	2549.71
5	801.05	152.41	241.73	327.56	3.83	413.56	157.13	2097.27
6	690.91	135.72	216.82	281.78	3.76	358.59	131.41	1818.99
7	599.02	125.46	198.78	248.64	3.72	322.92	116.41	1614.95
8	534.16	118.1	186.03	224.55	3.75	294.47	102.85	1463.91

- $f_{12}(\mathtt{d_w};\ 7)$, an i.i.d. random effect modelling the mean logBS at each day of week;
- $f_{13}(\mathtt{d_w},\ \mathtt{w_y};\ 7,\ 20)$, a tensor product spline, whose first margin is an i.i.d. random effect of $\mathtt{d_w}$ and the second margin is a cubic cyclic spline of $\mathtt{w_y}$. This component models the deviation of logBS seasonality at each day of week from the mean logBS seasonality f_1;
- $f_{14}(\mathtt{d_w},\ \{\mathtt{e_i},\ \mathtt{n_i}\};\ 7,\ 200)$, a tensor product spline, whose first margin is an i.i.d. random effect of $\mathtt{d_w}$ and the second margin is a rank 200 thin-plate spline of spatial coordinates. This component models the deviation of logBS over space at each day of week from the mean logBS over space f_2;
- $e_{\mathtt{id}}$, model error, assumed to be i.i.d. for model fitting.

Whether this model is reasonable or not is not of interest here. It is chosen for testing just to demonstrate that the new GAM fitting methods developed in this Chapter makes it feasible to estimate a daily logBS model. This test model involves 12460 coefficients for 387253 data.

Table 5.3 presents the fitting time for this model. Using 8 CPU core, model fitting completes in about 25 minutes. Figure 5.3 further illustrates parallel scaling for reduction, REML estimation and the overall GAM fitting. See figure caption for more details. In short, the parallel scaling is very good. (Model fitting time for model 3.6 using 8 CPU cores is also reported in Table 5.2.) In conclusion, the computational hurdle at the start of this Chapter is successfully overcome.

5.6 A comparison of computational capability with INLA

GAM estimation in `mgcv` is a type of frequentist shrinkage estimation or empirical Bayes estimation (see §1.1.5). Hierarchical Bayesian estimation treating smoothing parameters (or variance components as they are often known in literature of mixed models) as hyper parameters may be an alternative. The `R` package `INLA` using integrated nested Laplace approximation provides a practically efficient way for model estimation via this route.

Representing GAMs for logBS in `INLA` is not straightforward, as these models are rich in interaction. As an experiment, I began with an extremely simple GAM with only a single interaction term:

$$\mathtt{logBS_{id}} = f_1(\mathtt{w_y}) + f_2(\mathtt{d_w}) + f_3(\mathtt{d_w},\ \mathtt{w_y}) + f_4(\{\mathtt{e_i},\ \mathtt{n_i}\}) + \epsilon_{\mathtt{id}}.$$

The following explains how each function is constructed in `mgcv` and `INLA`.

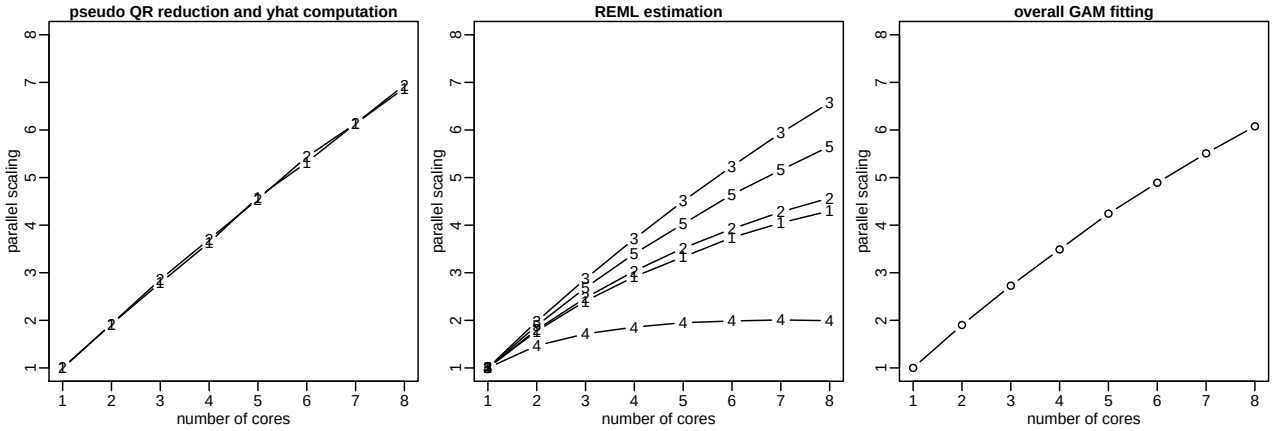


Figure 5.3: Illustration of parallel scaling in GAM fitting using methods developed in this Chapter. The **test model** assuming i.i.d. errors is fitted for experiment, and see Table 5.3 for the raw timing statistics used to compute parallel scaling factors in this figure. The panel on the left shows the parallel scaling for "fork-join" pseudo QR reduction (labelled with 1) and $\hat{\boldsymbol{y}}$ computation (labelled with 2). They have near equal parallel scaling and are both very impressive. The middle panel shows the parallel scaling for REML estimation, where lines labelled with 1 to 5 are respectively associated with the five computation steps in REML estimation, namely $\boxed{\log|\boldsymbol{S}_{\boldsymbol{\lambda}}|_+}$, $\boxed{\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}}$, $\boxed{\boldsymbol{V}_{\boldsymbol{\lambda}}}$, $\boxed{D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda})}$ and $\boxed{\log|\boldsymbol{H}_{\boldsymbol{\lambda}}|}$. The scaling property differs between these steps. $\boxed{\log|\boldsymbol{H}_{\boldsymbol{\lambda}}|}$ ("5") and $\boxed{\boldsymbol{V}_{\boldsymbol{\lambda}}}$ ("3") have very good scaling because these steps primarily involves matrix multiplications. $\boxed{\log|\boldsymbol{S}_{\boldsymbol{\lambda}}|_+}$ ("1") and $\boxed{\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}}$ ("2") scale slightly inferior, because they need to solve triangular system of linear equations apart from pivoted Cholesky factorization. $\boxed{D_p(\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}},\ \boldsymbol{\lambda})}$ ("4") scales poorly but this does not matter in practice, since this step takes almost negligible computation time (see for example, Table 5.3). The right panel shows the parallel scaling for overall GAM fitting, which is very good.

- In `mgcv`, f_1 is constructed as a cubic cyclic spline; in `INLA`, it is constructed as a first order random walk.
- f_2 is constructed as an i.i.d. random effect in both `mgcv` and `INLA`.
- f_3 is constructed as a tensor product spline in `mgcv` between a cubic cyclic spline margin and a random effect margin. In `INLA`, it is constructed as a Kronecker product model (see (Lindgren and Rue, 2015, §3.2)), where there is a first random walk of $\mathtt{w}_\mathtt{y}$ for each level of $\mathtt{d}_\mathtt{w}$.
- in `mgcv`, f_4 is constructed as a thin-plate spline; in `INLA`, it is constructed as a Gaussian random field with Matérn covariance, represented as a solution to a stochastic partial differential equation (SPDE) (Lindgren et al., 2011).

Fitting this model to all 387253 daily logBS in year 1967 only took 2 minutes in `mgcv` with less than 5 GB RAM, but still did not finish after 3 hours with `INLA`. In particular, `INLA` consumed all 128 GB RAM available on a E5-2650 v2 workstation and began to use increasingly more disk storage for swapping. I terminated the computation as the operating system was brought to a halt. There was then no need to try more complicated models with `INLA`. Fitting daily models with `INLA` is just infeasible.

Chapter 6

Preliminary modelling of logBS (a revisit)

Being able to fit a model as fast as possible makes many statistical questions answerable. For example, it is now possible to run cross-validation for model 3.6 and model 3.7 to see whether three-way interactions should be retained in a daily logBS model. In this Chapter, I will revisit daily logBS models from 1967 and annual mean logBS models.

6.1 A revisit to daily logBS model in 1967

6.1.1 Justifying three-way interactions with cross-validation

The cross-validation is performed as follows. Randomly partition the Monday logBS dataset (54386 data) into a training subset with 80% of the data and a test subset with the other 20%. Fit model 3.6 (without three-way interactions) and model 3.7 (with three-way interactions) to the training subset, and compute mean prediction squared error (MPSE) of both models on the test subset. Repeat this for 100 times, and compute the average MPSE as well as its 95%-confidence interval for both models. Table 6.1 presents the results. To conclude, these three-way interactions should be retained in the model.

6.1.2 A joint model for all days of week?

Model development so far for daily logBS in 1967 has been restricted to building separate models for each day of week. It is time to ask whether it is possible to build a joint model for all days of week. But a key question is: is this worthwhile? Extending model 3.7 to a joint model by allowing many splines to vary with $\texttt{d}_\texttt{w}$ can result in too many parameters, especially if those three-way interactions

Table 6.1: Cross-validation test for three-way interactions. "mean" gives the average MPSE over 100 simulations; "lwr" and "upr" respectively give lower and upper bound of the 95% confidence interval for the mean. There is strong evidence that three-way interactions should be retained in the model.

	mean	lwr	upr
model 3.6	0.2289	0.2280	0.2298
model 3.7	0.2167	0.2158	0.2176

are extended to four-way interactions.

Let us first attempt the following test model:

joint model

$$\begin{aligned}\texttt{logBS}_{\texttt{id}} = {} & f_0(\texttt{E}_\texttt{i};\ 5) + f_1(\texttt{w}_\texttt{y};\ 10) + f_2(\{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\};\ 150) + f_3(\texttt{w}_\texttt{y},\ \{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\};\ 20,\ 150) + \\ & f_4(\texttt{T}^0_\texttt{id};\ 15) + f_5(\texttt{T}^0_\texttt{id},\ \{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\};\ 10,\ 100) + \\ & f_6(\texttt{T}^*_\texttt{id};\ 15) + f_7(\texttt{T}^*_\texttt{id},\ \{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\};\ 10,\ 100) + \\ & f_8(\texttt{T}^0_\texttt{id},\ \texttt{T}^*_\texttt{id};\ 15,\ 15) + f_9(\texttt{i};\ 1275) + f_{10}(\texttt{w}_\texttt{y};\ 52) + f_{11}(\texttt{h}_\texttt{i};\ 5) + \\ & f_{12}(\texttt{w}_\texttt{y},\ \texttt{T}^0_\texttt{id},\ \{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\};\ 10,\ 10,\ 30) + f_{13}(\texttt{w}_\texttt{y},\ \texttt{T}^*_\texttt{id},\ \{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\};\ 10,\ 10,\ 30) + \\ & f_{14}(\texttt{d}_\texttt{w};\ 7) + f_{15}(\texttt{d}_\texttt{w},\ \texttt{w}_\texttt{y};\ 7,\ 20) + f_{16}(\texttt{d}_\texttt{w},\ \{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\};\ 7,\ 150) + \\ & f_{17}(\texttt{d}_\texttt{w},\ \texttt{T}^0_\texttt{id};\ 7,\ 10) + f_{18}(\texttt{d}_\texttt{w},\ \texttt{T}^0_\texttt{id};\ 7,\ 10) + f_{19}(\texttt{d}_\texttt{w},\ \texttt{w}_\texttt{y},\ \{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\};\ 7,\ 10,\ 50) + e_{\texttt{id}}.\end{aligned}$$

New components to model 3.7 are

- $f_{14}(\texttt{d}_\texttt{w};\ 7)$, an i.i.d. random effect modelling the mean logBS at each day of week;
- $f_{15}(\texttt{d}_\texttt{w},\ \texttt{w}_\texttt{y};\ 7,\ 20)$, a tensor product spline, whose first margin is an i.i.d. random effect of $\texttt{d}_\texttt{w}$ and the second margin is a cubic cyclic spline of $\texttt{w}_\texttt{y}$. This component models the deviation of logBS seasonality at each day of week from the mean logBS seasonality f_1;
- $f_{16}(\texttt{d}_\texttt{w},\ \{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\};\ 7,\ 200)$, a tensor product spline, whose first margin is an i.i.d. random effect of $\texttt{d}_\texttt{w}$ and the second margin is a rank 200 thin-plate spline of spatial coordinates. This component models the deviation of logBS over space at each day of week from the mean logBS over space f_2;
- $f_{17}(\texttt{d}_\texttt{w},\ \texttt{T}^0_\texttt{id};\ 7,\ 10)$, a tensor product spline, whose first margin is an i.i.d. random effect of $\texttt{d}_\texttt{w}$ and the second margin is a cubic regression spline of $\texttt{T}^0_\texttt{id}$. This component models how the average effect of daily minimum temperature f_4 may vary within a week;
- $f_{18}(\texttt{d}_\texttt{w},\ \texttt{T}^*_\texttt{id};\ 7,\ 10)$, a similar component to f_{17} associated with diurnal temperature difference;
- $f_{19}(\texttt{d}_\texttt{w},\ \texttt{w}_\texttt{y},\ \{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\};\ 7,\ 10,\ 50)$, a three-margin tensor product spline modelling how the spatially varying seasonality f_3 can further vary within a week;
- $e_{\texttt{id}}$, model error. This is definitely not i.i.d.. An appropriate assumption may be proposed after examining residuals of this model fitted under i.i.d. error assumption.

Note that the k value for the spatial margin in f_{19} is deliberately kept low to avoid ending up with too many parameters. Still, this model has 14394 regression coefficients.

A key concern is how good this joint model is in modelling space-time relation. It is straightforward to produce variograms of its spatial residuals on all 365 days, but it is probably sufficient to just produce such variograms at all 52 Mondays, so that a comparison can be made with Figure 3.46 from the fitted Monday-only model. This is done in 6.1, and it turns out that the joint model is very inadequate in spatial modelling. Just looking at the first row, residuals of the joint model have unmodelled spatial autocorrelation on day 2, 16, 23, 30, while residuals of the Monday-only model on these days are all fine.

The model may be improved if f_{12} and f_{13} are extended to four-way interactions. However, this will raise the number of coefficients to 39450. Even if there is no computational difficulty, why should we fit this model at all? If all components of model 3.7 should vary with day of week, then why not just build seven separate models?

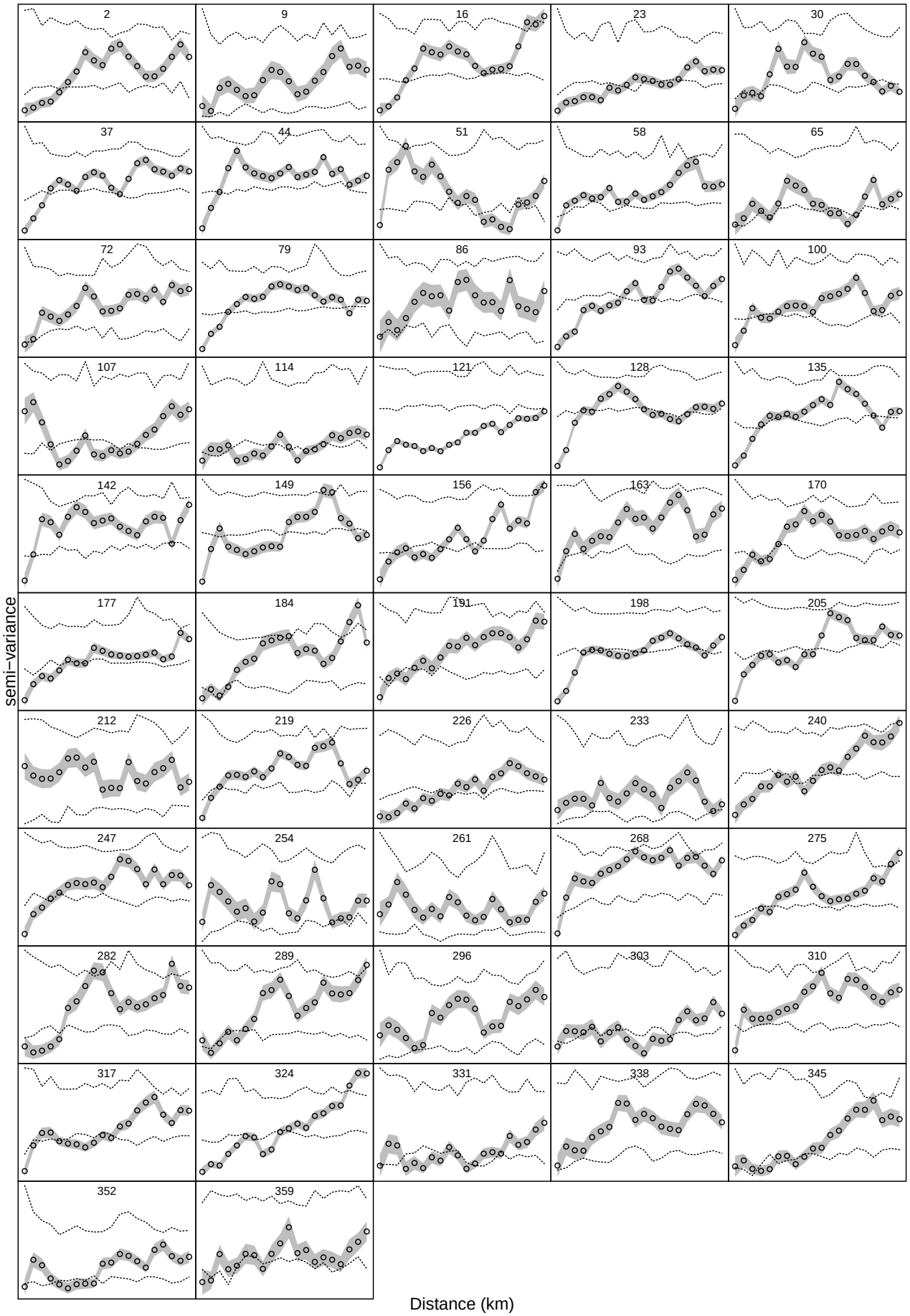


Figure 6.1: Empirical variograms of spatial residuals from fitted joint model on each of the 52 Mondays in year 1967. Compared with Figure 3.46 from fitted Monday-only model, the joint model is very inadequate in spatial model.

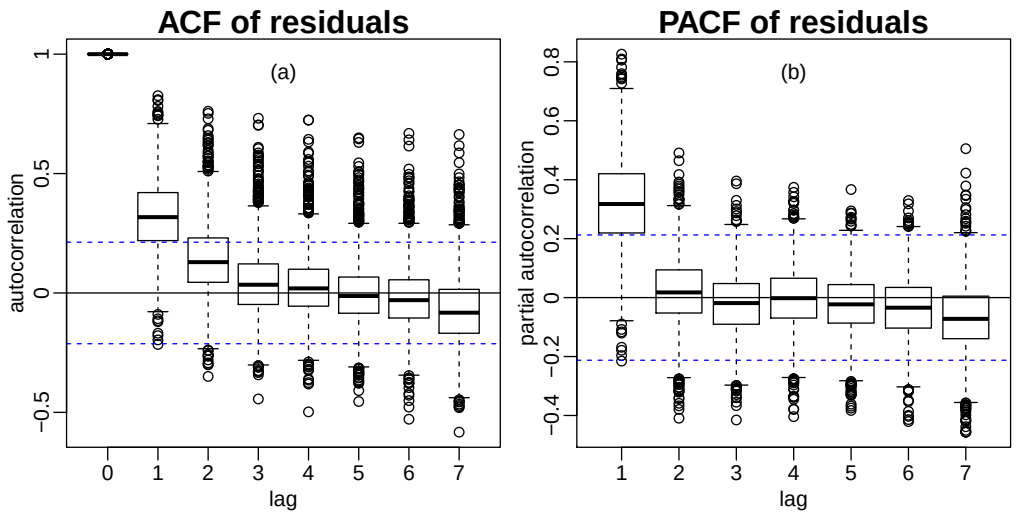


Figure 6.2: ACF (panel (a)) and PACF (panel (b)) of residuals from all stations in fitted joint model (assuming i.i.d. errors). The results are presented in boxplots, and the blue dashed lines are "95% confidence interval" computed based on the median sample size over all stations. The cutoff of PACF at lag 1 shows that on average, residuals from each station are AR(1) correlated.

Building separate models for each day of week does offer some convenience in practice. On one hand, temporal autocorrelation won't step in model development; on the other hand, the number of data used for model fitting is always 1 / 7 of the volume of the complete dataset hence model fitting is always faster.

However, as part of methodology development, it is useful to demonstrate how temporal autocorrelation may be dealt with in a model for daily data. In the next section, I will explain the construction of block AR(1) errors where the AR(1) coefficient can be estimated by the golden-section search method previously covered. And in the next next section I will give some practical tricks that help speed up the golden-section search when fitting GAMs for large datasets.

6.1.3 Block AR(1) errors with autocorrelation coefficient ω

Fitting the joint model assuming i.i.d. error leaves unmodelled temporal autocorrelation in residuals, as panel (a) of Figure 6.2 shows. Panel (b) also shows that the average pattern of such autocorrelation can be modelled by an AR(1) process. In reality, the autocorrelation coefficient varies greatly across stations (and I have previously demonstrated this in the histogram from Figure 3.24), but estimating all these individual coefficients is not practically feasible. For example, the golden-section search method introduced in §2.2.3 can only deal with a single unknown coefficient. So here the simplest strategy is taken, by only modelling the average autocorrelation coefficient. In other words, errors $e_{\mathtt{i},\mathtt{d}}$ from station `i` are an AR(1) process

$$e_{\mathtt{i},\mathtt{d}} = \omega \cdot e_{\mathtt{i},\mathtt{d}-1} + \epsilon_{\mathtt{i},\mathtt{d}-1},$$

where the autocorrelation coefficient ω is common for all stations. This produces a block diagonal weight matrix for the resulting additive model, where each diagonal block is a symmetric tri-diagonal matrix like (2.1). For example, if there are two AR(1) processes, each of length 4, then the weight matrix is

$$\boldsymbol{W} = \left(\begin{array}{cccccccc} A & B & & & & & & \\ B & C & B & & & & & \\ & B & C & B & & & & \\ & & B & A & 0 & & & \\ & & & 0 & A & B & & \\ & & & & B & C & B & \\ & & & & & B & C & B \\ & & & & & & B & A \end{array}\right), \tag{6.1}$$

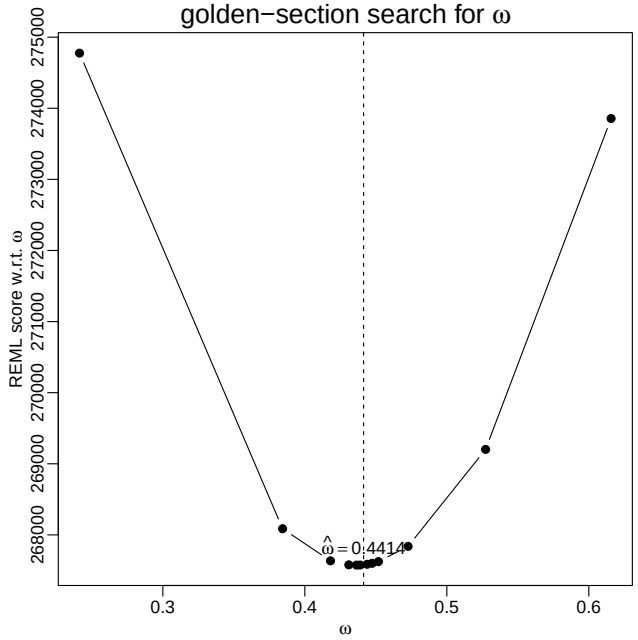


Figure 6.3: Illustration of golden-section search in estimation of ω for the joint model. It converges in 14 steps, giving a point estimate $\hat{\omega} = 0.4414$.

where $A = 1/(1-\omega^2)$, $B = -\omega A$ and $C = (1+\omega^2)A$. There are two ways to store this matrix: storing each diagonal separately or storing its non-zero elements by row. Either way, elements from the lower or upper sub-diagonal need not be stored due to symmetry (this explains why the lower sub-diagonal in the above matrix is coloured grey). For standardization (see §2.2.2 if you need a revision), the upper triangular Cholesky factor of this weight matrix is

$$\boldsymbol{W}^{\frac{1}{2}} = \left(\begin{array}{cccccccc} D & E & & & & & & \\ & D & E & & & & & \\ & & D & E & & & & \\ & & & 1 & 0 & & & \\ \hline & & & & D & E & & \\ & & & & & D & E & \\ & & & & & & D & E \\ & & & & & & & 1 \end{array}\right),$$

where $D = \sqrt{A}$ and $E = -\omega\sqrt{A}$. To estimate ω via golden-section search, the log-determinant of the weight matrix is required for evaluation of REML score. The block structure makes this computation very easy, as the determinant (or log-determinant) of the full weight matrix is just the product (or sum) of the determinant (or log-determinant) for each block. In §2.2.2 I have shown that the log-determinant of a length-$l_{\mathtt{i}}$ AR(1) process is $-(l_{\mathtt{i}}-1)\log(1-\omega^2)$, thus it is straightforward to see that $\log|\boldsymbol{W}| = -(n-\gamma)\log(1-\omega^2)$, where γ is the number of AR(1) blocks and $n = \sum_{\mathtt{i}} l_{\mathtt{i}}$ is the total number of data from all blocks.

Figure 6.3 illustrates the golden-section search for the joint model. It converges in 14 steps, giving a point estimate $\hat{\omega} = 0.4414$. Figure 6.4 sketches the ACF and PACF of model residuals, as well as the ACF of standardized residuals. It can be seen that on average, residual autocorrelation are eliminated.

6.1.4 Speeding up golden-section search

Golden-section search makes GAM fitting substantially more costly. With 14 steps for convergence, it is essentially fitting the joint model for 14 times, so the overall fitting time becomes as long as 7 hours. In practice, the fitting process can be slightly speeded up by specifying a narrower search interval.

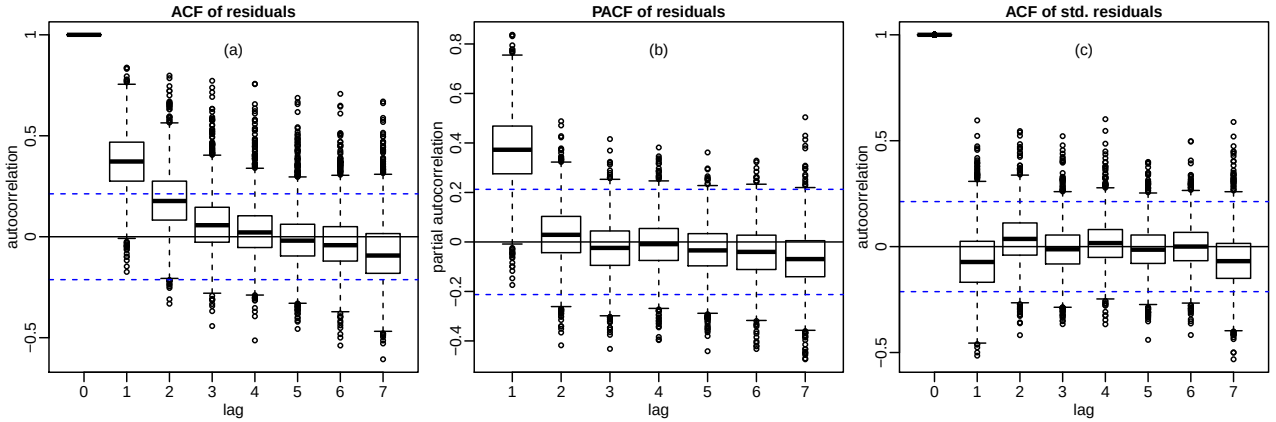


Figure 6.4: ACF (panel (a)) and PACF (panel (b)) of residuals from all stations in fitted joint model (assuming AR(1) errors). Panel (c) further gives the ACF of standardized residuals. The results are presented in boxplots, and the blue dashed lines are "95% confidence interval" computed based on the median sample size over all stations. The AR(1) assumption for mean errors over all stations is adequate, and on average, standardized residuals at each station are uncorrelated.

Previously a very general interval (0.01, 0.99) has been used, now we may restrict it to, for example (0.3, 0.5). As a result, the search converges in 10 steps. This reasonable guess is made by spotting Figure 6.2, where the mean lag-1 autocorrelation is slightly above 0.3. This value is always lower than the one to be estimated via golden-section search, because some proportion of autocorrelation has been fitted by splines under i.i.d. error assumption. So it is safe to use 0.3 as the lower bound of the search interval. 0.5 is chosen as the upper bound because this value is readily much greater than the 75% quantile of lag-1 autocorrelation.

Another way to speed up convergence is to use a slightly higher numerical tolerance (see Figure 2.12) for convergence test. Previously $\epsilon = 0.001$ has been used, but this is probably more than necessary. In particular, since the objective function is flat near its minimum (the first derivative is close to 0), search steps are increasingly smaller near minimum and the decrease in function value is insignificant. There isn't much point spending hours on this tiny improvment. Increasing the tolerance to 0.005 requires only 6 steps for convergence, given a point estimate at 0.4413 which is basically no different from 0.4414.

A even better way for faster convergence is to use an adaptive tolerance for REML estimation (see ϵ in Figure 4.9). Previously for each trial ω from the outer golden-section loop, 17 Newton-Raphson steps are taken to achieve convergence subject to $\epsilon = 10^{-6}$. Think about this carefully, there is little sense to iterate Newton-Raphson to covergence with such a high precision before the golden-section has a sign for convergence. Of course, while this is a reasonable idea, practically programming this needs some care to ensure stability of the algorithm. At the moment I haven't implemented this idea.

6.1.5 Summary and visualization of seven separate models

The seven fitted models have already been summarized in Table 3.10. I will now show some diagnostic plots for these models.

Figures 6.5, 6.6 and 6.7 respectively check i.i.d. assumption for model errors, station-specific random effect and week of year random effect. No violation is seen. Figure 6.8 sketches residuals against fitted values on each day of week. I am not surprised that there seems to be some mean-variance dependence at low fitted values and high fitted values; this is just a reflection of the difficulty to model extreme values in daily logBS. Overall, there is no violation of constant error variance assumption.

Without a joint model it is less straightforward to compare components between different models.

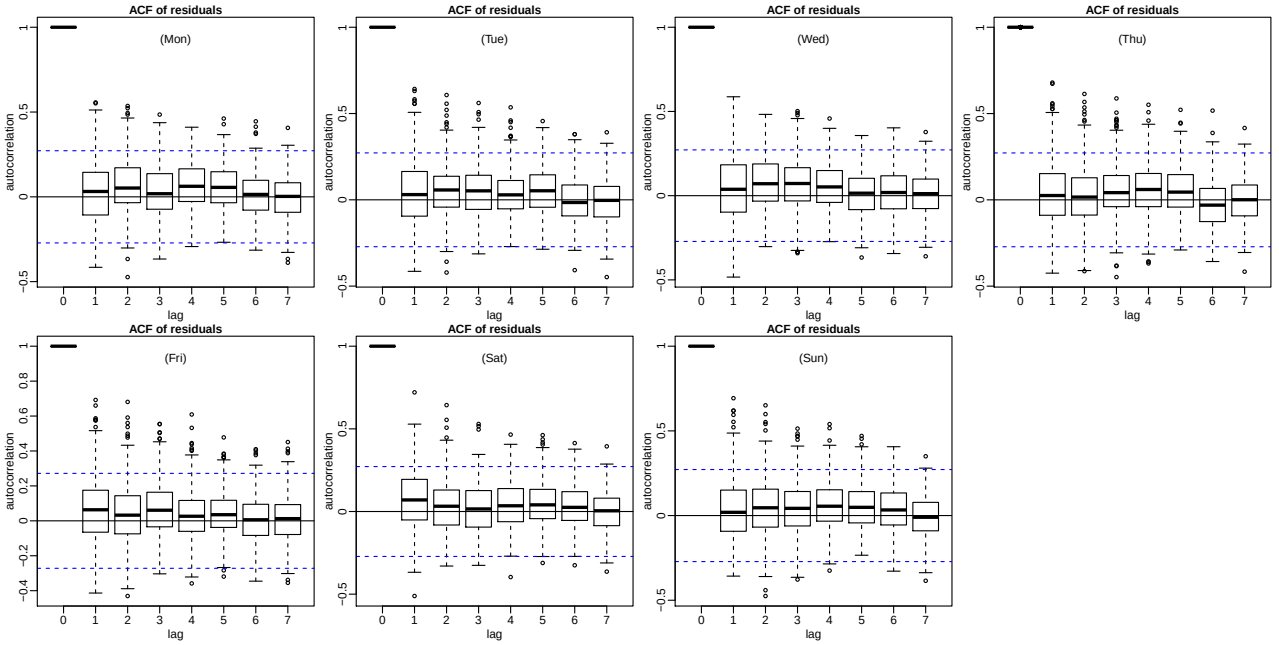


Figure 6.5: ACF of residuals of the fitted model 3.6 for each day of week. Results are presented in boxplots, and the blue dashed lines are "95% confidence interval" computed based on the median sample size over all stations. It is evident that on average, there is no temporal autocorrelation in residuals, so the i.i.d. model error assumption is justified.

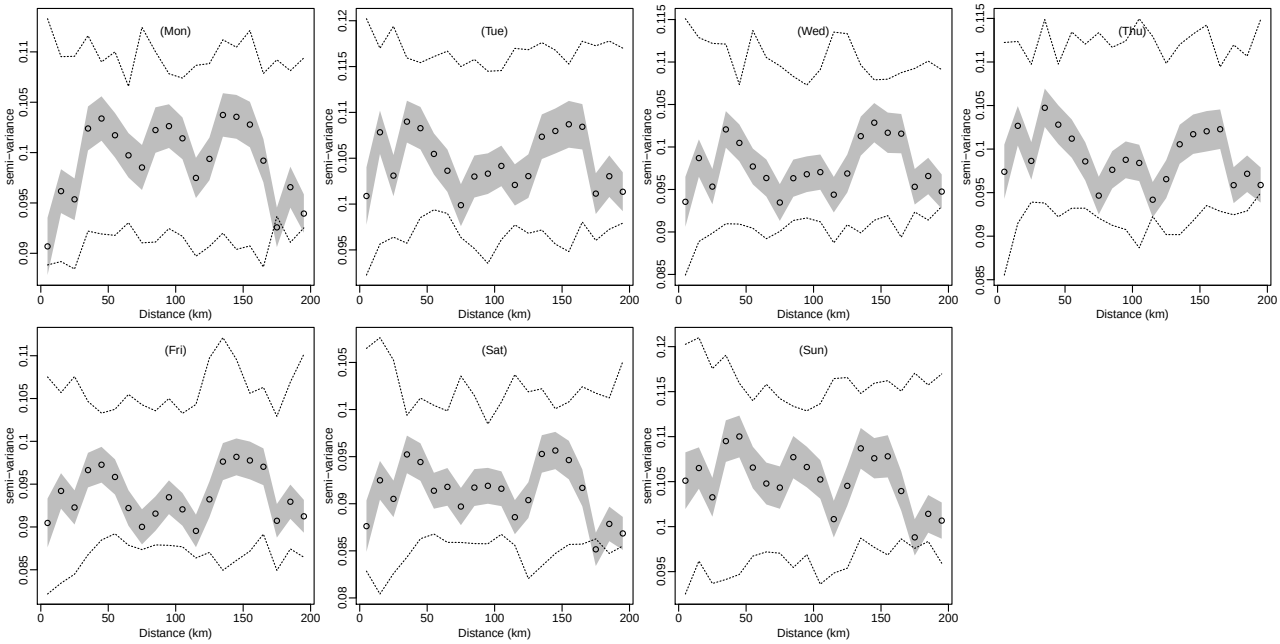


Figure 6.6: Variogram of station-specific i.i.d. random effect $\hat{f}_9$ in the fitted model 3.6 for each day of week. No spatial autocorrelation is spotted, thus the i.i.d. assumption is validated.

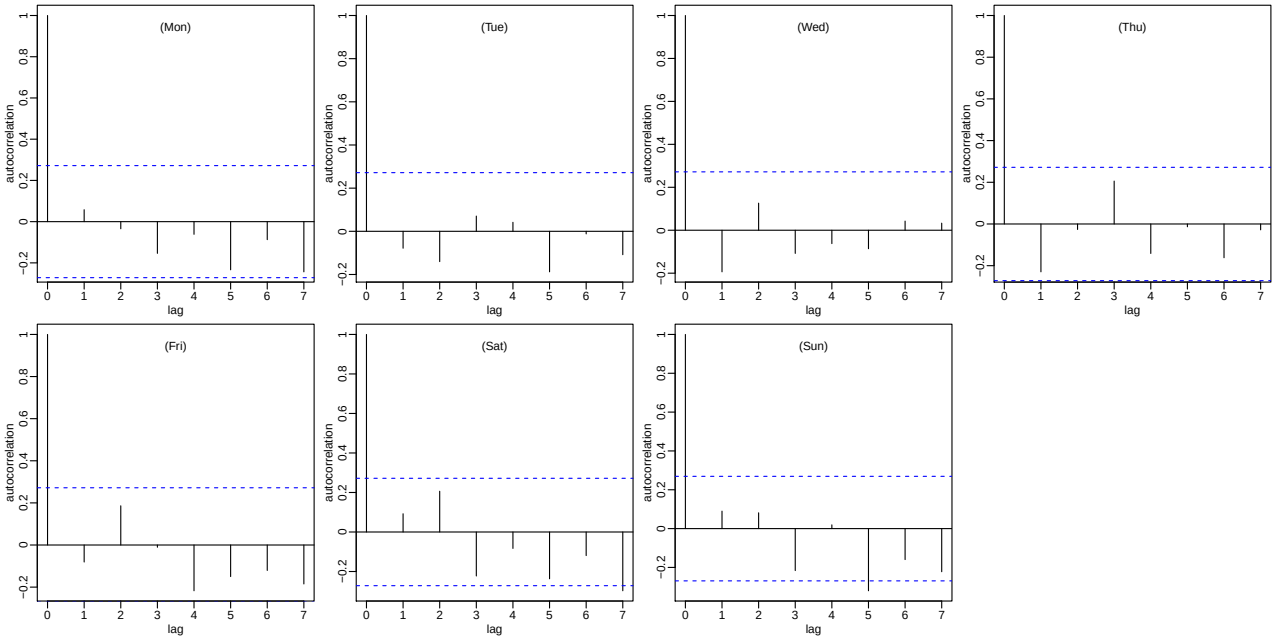


Figure 6.7: ACF of week of year i.i.d. random effect $\hat{f}_{10}$ in the fitted model 3.6 for each day of week. No temporal autocorrelation is spotted, thus the i.i.d. assumption is not violated.

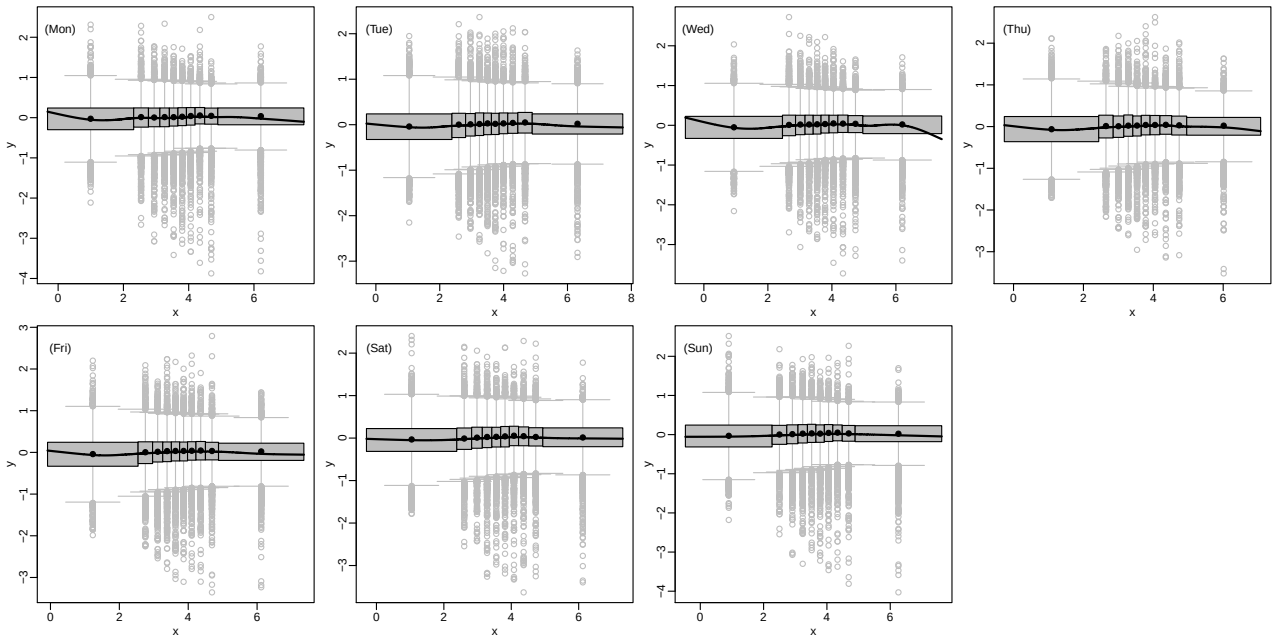


Figure 6.8: Residuals against fitted values for each day of week. I am not surprised that there seems to be some mean-variance dependence at low fitted values and high fitted values; this is just a reflection of the difficulty to model extreme values in daily logBS. Overall, there is no violation of constant error variance assumption.

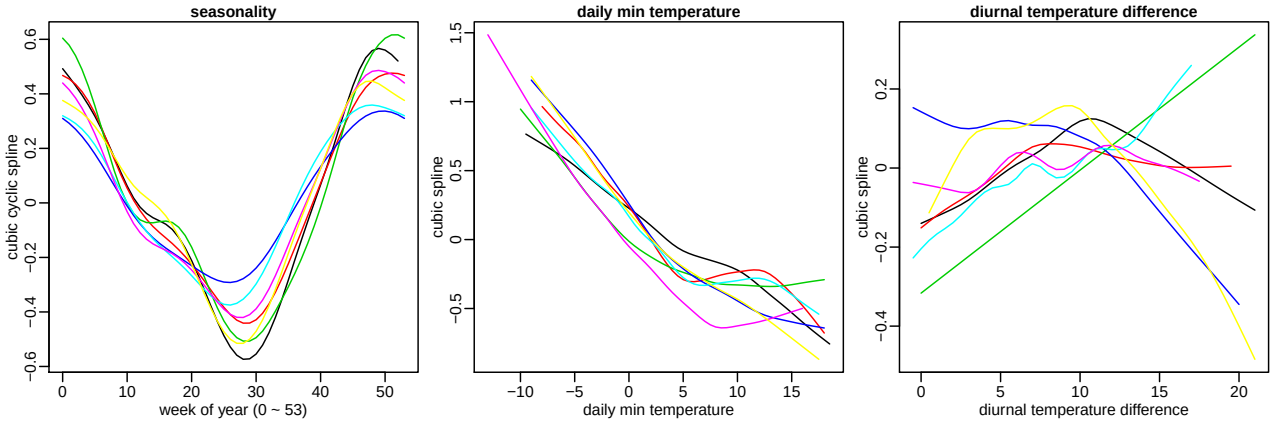


Figure 6.9: Estimated seasonality and main temperature effects from different models (black: Monday; red: Tuesday; green: Wednesday; blue: Thursday; cyan: Friday; magenta: Saturday; yellow: Sunday). Note that these curves are not comparable in values, as different models have different intercepts and those curves are subject to different centring constraints. Only their shapes can be compared.

It is also difficult to assess weekly effects. So Figure 6.9 comparing estimated seasonality and main temperature effects from different models is very informal. The only useful message is probably that these effects have a different shape on each day of week. The effects of diurnal temperature difference is very difficult to interpret. I think this once again signals that more daily covariates are needed for building models, otherwise relationship between logBS and those unavailable daily variables have to be described by sophisticated relationship between logBS and temperature.

6.1.6 Summary

In this section I revisited the spatial-temporal modelling of daily logBS in year 1967, completing the previously unfinished discussion regarding three-way interactions between space, time and temperature. The cross-validation result supports the inclusion of these components in a daily model. However, the presence of such three-way interactions makes it difficult to extend separate models per day of week to a joint model. A relatively modest joint model without extending those three-way interactions to four-way interactions was attempted, but it turned out inadequate in spatial modelling. It was then concluded that building an adequate joint model would result in a model with too high complexity, so I finally stayed with the strategy of building separate models.

Building separate models for each day of week eliminates the needs for modelling temporal autocorrelation in daily data, but I still discussed how such autocorrelation can be dealt with, by assuming model errors at each station to be AR(1) with the same correlation coefficient ω. This assumption makes estimation of ω straightforward via golden-section search. Perhaps the assumption of a common ω is over restricted. In principle it is farily easy to assume a different $\omega_{\mathtt{i}}$ for each station: the resulting weight matrix is still symmetric tri-diagonal, with $A_{\mathtt{i}}$, $B_{\mathtt{i}}$ and $C_{\mathtt{i}}$ in place of A, B and C. However, this ends up with too many parameters in the weight matrix and how to estimate so many $\omega_{\mathtt{i}}$'s? A probably more moderate strategy is to assume that ω_i varies smoothly over space. In fact, thinking toward this direction suggests that $\boldsymbol{W}^{-1}\phi$ be constructed as a spatial-temporal covariance matrix. The link can be established as follows. A closer look at (6.1) reveals that the weight matrix is just the Kronecker product ("$\otimes$") beween an identity matrix $\boldsymbol{I}$ and the inverse correlation matrix of an AR(1) process $\boldsymbol{W_t}$ (as in (2.1)). If $\boldsymbol{I}$ is thought of as the inverse correlation of an i.i.d. spatial process, then it is certainly natural to generalize it to an inverse spatial correlation matrix $\boldsymbol{W_s}$ and construct $\boldsymbol{W} = \boldsymbol{W_s} \otimes \boldsymbol{W_t}$. The complexity of $\boldsymbol{W}$ is well under control, because $\boldsymbol{W_t}$ just has a single parameter ω and $\boldsymbol{W_s}$ often just has a single parameter as well. However, to make practical computation (as well as storage) feasible, $\boldsymbol{W_s}$ has to be very sparse. Implementation of this in GAMs could be an interesting topic to pursue in future.

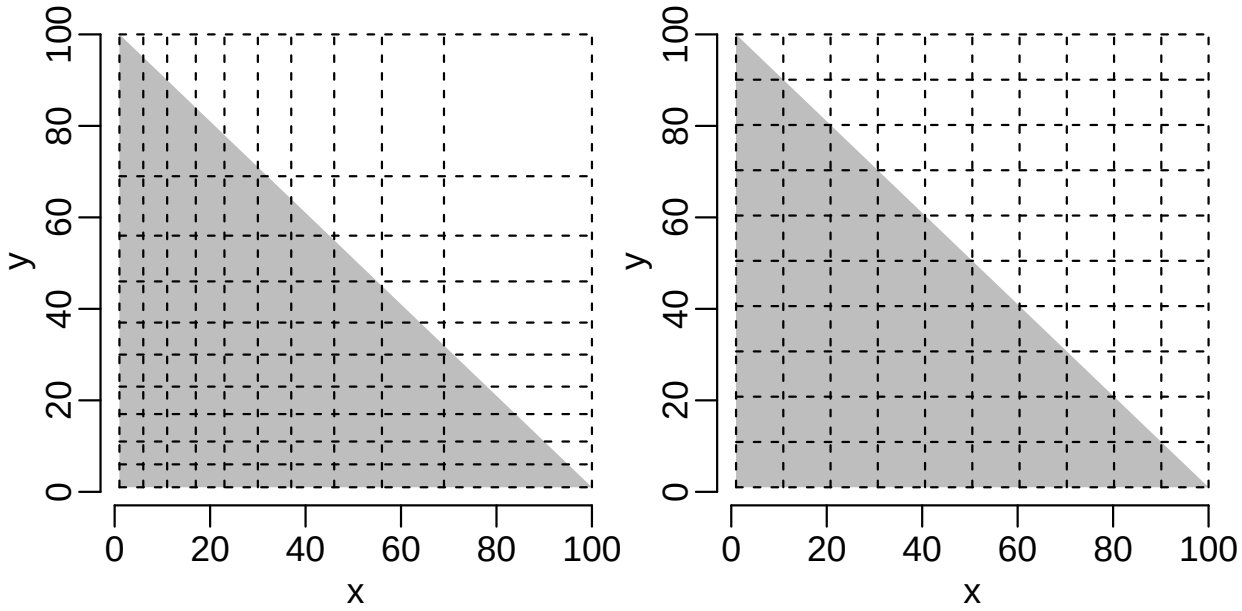


Figure 6.10: A bivariate tensor product spline example to illustrate different knots placement strategies. Suppose we want to estimate a tensor product spline $s(x, y)$ defined on a $[1, 100] \times [1, 100]$ square, but data are only sampled at grid points $(i,\ j)$, $i,\ j = 1,\ 2,\ \cdots,\ 100$ in the lower triangular domain (the gray shaded area). If both margins are modelled by a cubic regression spline with 11 knots, the left panel places knots at 0%, 10%, ..., 100% quantiles on each margin, while the right panel places knots evenly on each margin. In both panels, intersections of horizontal and vertical dashed lines give the resulting knots of the tensor product spline.

6.2 A revisit to annual mean logBS model

Previously in §3.4.6 I have realized the extrapolation problem in making spatial-prediction as the Network became sparser and sparser. Now I am going make an attempt to address this problem. My attempt turned out unsuccessful, but it is still interesting to present it here.

From Figure 3.34 it is obvious that the tensor product spline $f_3(\texttt{y},\ \{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\})$ modelling the space-time interaction is the cause of extrapolation. A conjecture is that the thin-plate spline margin is overflexible in later years when there were insufficient spatial information. So I began to think about a way to restrict flexibility of this margin.

The issue may be related to knots placement. To motivate this, consider a bivariate tensor product spline example illustrated in Figure 6.10. Knots placement in the left panel is more sensible, as knots are more densely distributed in the sampling region. Fewer knots in the unsampled region means less risk of extrapolation over there. Analogy can be made for $f_3(\texttt{y},\ \{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\})$. Since I am to model space-time interaction with a tensor product spline, knots should probably be placed in space-time coordinates where information of such interaction is rich. In other words, knots of the thin-plate spline should be placed at spatial locations where long-term Black Smoke sampling is available. However, the automatic knots placement for a thin-plate spline in `mgcv` disregards this, ending up using all unique 2626 site locations. Had all stations had an adequately long monitoring history, such auto knots placement would not be problematic, but stations of the Black Smoke Network generally had a very short life (see Figure 3.25). Therefore, such auto knots placement may result in unreasonable spatial-temporal extrapolation.

To obtain a sensible set of knots, consider those 5km $\times$ 5km pixels of the spatial prediction domain (see Figure 3.1). Among all 3949 pixels, 107 have observations for at least 30 years and 61 have observations for at least 35 years. Panels (a) and (b) of Figure 6.11 respectively pinpoint these pixels, which are respectively suitable as knots of a rank-107 and a rank-61 thin-plate spline.

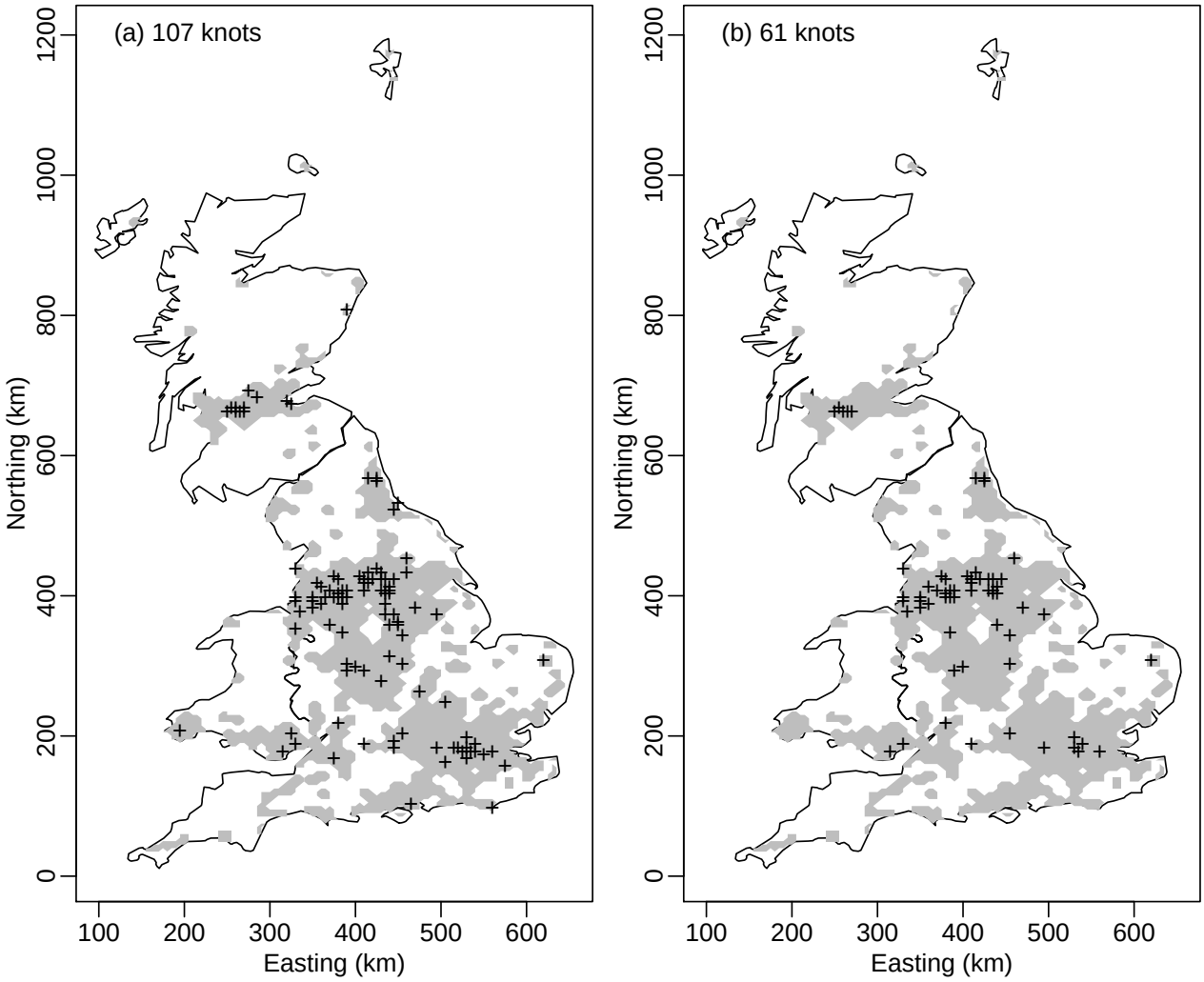


Figure 6.11: Attempts to mitigate spatial-temporal extrapolation by placing knots of the thin-plate spline at spatial locations where long-term Black Smoke sampling is available. Panels (a) and (b) respectively select 125, 75 and 30 pixels from the spatial domain (see Figure 3.1) as knots, where at least 30 and 35 years' observations are available.

Table 6.2: Cross-validation for different knots placement strategies for thin-plate spline margin of f_3 in model 3.5. Column "mean" gives average MPSE from 100 simulations; "lwr" and "upr" respectively give lower and upper bound of 95%-confidence interval of "mean".

	$m=1$			$m=2$			$m=3$		
	mean	lwr	upr	mean	lwr	upr	mean	lwr	upr
auto	0.6282	0.6238	0.6326	0.6255	0.6216	0.6294	0.5856	0.5824	0.5888
panel (a)	0.6266	0.6220	0.6311	0.6264	0.6228	0.6300	0.5966	0.5933	0.5998
panel (b)	0.6151	0.6106	0.6195	0.6137	0.6102	0.6171	0.5824	0.5791	0.5856
	$m=4$			$m=5$					
	mean	lwr	upr	mean	lwr	upr			
auto	0.5745	0.5706	0.5784	0.5668	0.5633	0.5704			
panel (a)	0.5851	0.5808	0.5894	0.5742	0.5706	0.5779			
panel (b)	0.5758	0.5717	0.5798	0.5671	0.5633	0.5709			

To justify such manual knots placement, I decided to do cross-validation which is designed as follows. Randomly select 1000 stations and delete their last m years' observations. Use the deleted data as test dataset and the rest of the data as training dataset. Fit model 3.5 to the training dataset, with three different strategies for knots placement: auto, panel (a), panel (b), then compute the mean prediction squared error on test dataset. The simulation is repeated for 100 times. In addition, I will try $m = 1,\ 2,\ 3,\ 4,\ 5$. Table 6.2 presents the results of cross-validation. It is hard to draw a firm

conclusion from those results. Strategy "panel (a)" almost always gives higher prediction error than "auto", except for $m = 1$ case. Then strategy "panel (b)" gives lower prediction error than "auto" for $m = 1,\ 2,\ 3$, but the prediction error is then higher "auto" for $m = 4,\ 5$. So there is no solid evidence that manual knots placement yields a better model. Unfortunately I have to admit that I can not resolve this issue in this thesis.

Chapter 7

A model for all daily logBS?

Plenty of logBS models have been built by now, but only a small proportion of the full daily logBS dataset has been used. What if I am to fit a model using the complete dataset? Is this computationally feasible?

7.1 A test model

The following is a test model for daily logBS on Monday:

Model 7.1:

$$\begin{aligned}\texttt{logBS}_{\texttt{id}} = {} & f_0(\texttt{E}_\texttt{i};\ 5) + f_1(\texttt{h}_\texttt{i};\ 5) + f_2(\{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\};\ 150) + f_3(\texttt{i};\ 2874) + \\ & f_4(\texttt{w};\ 40) + f_5(\texttt{w};\ 2310) + f_6(\texttt{E}_\texttt{i},\ \texttt{w};\ 5,\ 40) + f_7(\texttt{w},\ \{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\};\ 40,\ 150) + \\ & f_8(\texttt{w}_\texttt{y};\ 10) + f_9(\texttt{w}_\texttt{y},\ \{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\};\ 20,\ 150) + f_{10}(\texttt{w}_\texttt{y},\ \texttt{w};\ 20,\ 40) + \\ & f_{11}(\texttt{T}^0_\texttt{id};\ 15) + f_{12}(\texttt{T}^*_\texttt{id};\ 15) + f_{13}(\texttt{T}^0_\texttt{id},\ \texttt{T}^*_\texttt{id};\ 15,\ 15) + \\ & f_{14}(\texttt{T}^0_\texttt{id},\ \{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\};\ 10,\ 100) + f_{15}(\texttt{T}^*_\texttt{id},\ \{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\};\ 10,\ 100) + \\ & f_{16}(\texttt{w}_\texttt{y},\ \texttt{T}^0_\texttt{id},\ \{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\};\ 10,\ 10,\ 30) + f_{17}(\texttt{w}_\texttt{y},\ \texttt{T}^*_\texttt{id},\ \{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\};\ 10,\ 10,\ 30) + \\ & f_{18}(\texttt{T}^0_\texttt{id},\ \texttt{w};\ 10,\ 40) + f_{19}(\texttt{T}^*_\texttt{id},\ \texttt{w};\ 10,\ 40) + f_{20}(\texttt{r}_\texttt{im};\ 10) + \epsilon_{\texttt{iw}}.\end{aligned}$$

- f_0 is a random effect modelling the mean of logBS from stations of environmental type $\texttt{E}_\texttt{i}$ (5 types in total);
- f_1 is a natural cubic spline of $\texttt{h}_\texttt{i}$ with 5 knots;
- f_2 is a rank-150 thin-plate spline modelling the general spatial distribution of logBS across stations (invariant with time);
- f_3 is an i.i.d. random effect for station $\texttt{i}$ (2874 stations in total);
- f_4 is a natural cubic splin modelling the general long term trend (common to all stations);
- f_5 is an i.i.d. random effect for $\texttt{w}$ (2310 weeks in total);
- f_6 is a tensor product spline, whose first margin is a random effect for $\texttt{E}_\texttt{i}$ and the second margin is cubic spline of $\texttt{w}$. This component models how f_4 varies for each $\texttt{E}_\texttt{i}$;
- f_7 is a tensor product spline, whose first margin is a natural cubic spline of $\texttt{w}$ and the second margin is a thin-plate spline of $\{\texttt{e}_\texttt{i},\ \texttt{n}_\texttt{i}\}$. This component models how f_4 varies over space;

- f_8 is a cubic cyclic spline of $\mathtt{w_y}$ with 20 knots, modelling the mean seasonality of logBS at all stations;
- f_9 is a tensor product spline whose first margin is a cubic cyclic spline of $\mathtt{w_y}$ and the second margin is a thin-plate spline of $\{\mathtt{e_i},\ \mathtt{n_i}\}$. This component models how f_8 varies over space;
- f_{10} is a tensor product spline whose first margin is a cubic cyclic spline of $\mathtt{w_y}$ and the second margin is a natural cubic spline of $\mathtt{w}$. This component models how f_8 varies in a long term;
- f_{11} is a natural cubic spline with 10 knots for daily minimum temperature, modelling the mean effect of this variable at all stations;
- f_{12} is a similar component to f_{11}, associated with diurnal temperature difference;
- f_{13} is a tensor product spline modelling the interaction between two temperature variables;
- f_{14} is a tensor product spline whose first margin is a natural cubic spline of $\mathtt{T^0_{id}}$ and the second margin is a thin-plate spline of $\{\mathtt{e_i},\ \mathtt{n_i}\}$. This component models how f_{11} varies over space;
- f_{15} is a similar component to f_{14}, associated with diurnal temperature difference;
- f_{16} and f_{17} are three-way interactions (three-margin tensor product splines) between space, time and temperature. These components model how f_{14} and f_{15} vary within a year;
- f_{18} and f_{19} are two-margin tensor product splines, modelling how f_{11} and f_{12} vary in a long term;
- f_{20} is natural cubic spline of monthly rainfall, modelling the mean effect of this variable at all stations in all time;
- $\epsilon_{\mathtt{iw}}$ is an i.i.d. model error.

This model is proposed based on previous experience on logBS modelling.

- Components f_0 to f_7 are motivated by model 3.5 for annual mean logBS, except that time variable year $\mathtt{y}$ is replaced by week $\mathtt{w}$, motivated by model 3.1 in *Manchester 11* case study.
- Components f_8 to f_{13} are also motivated by *Manchester 11* case study, where it was shown that time-varying seasonality could be modelled by an interaction between $\mathtt{w_y}$ and $\mathtt{w}$ and that temperature variables are useful for predicting logBS;
- Components f_{14} to f_{17} are motivated by model 3.7 for Monday logBS in 1967, where is was shown that temperature effects would vary over space and time;
- Components f_{18} and f_{19} are added to model how relationship between logBS and temperature might change over decades (In §3.2.6 I mentioned that such relationship can not be estimated from *Manchester 11* case study due to small sample size);
- f_{20} is added for monthly rainfall. This variable was not included in any previous models, because the relationship between daily logBS and monthly rainfall appeared very weak (see for example Figures 3.13 and 3.40). However, adding a low-rank cubic spline for rainfall does not make model estimation any more difficult; if this variable indeed has no effect, this spline may be "penalized out".

Note that I can go on adding more components. For example, by allowing f_{16} and f_{17} to vary with $\mathtt{w}$, or by allowing f_{18} and f_{19} to vary with space. However, this model already has 22571 regression coefficients. There are in total 1342159 Monday logBS observations, so on average, there is one coefficient per 59.5 data.

This test model may be useful for two purposes.

1. The difficulty of modelling daily logBS can be assessed. The test model is readily flexible enough, and if it turns out inadequate, it is very likely that logBS can not be properly modelled using GAMs and there is no further point to go on building a joint model for all days of week.

2. It serves a very good example for investigating GAM computation methods. More than half of the model components are tensor product splines and there are even three-margin tensor product splines. Fitting such model is not a trivial task. Questions may be raised on whether the GAM computation methods developed in Chapter 5 should be further optimized to accommodate such complicated models.

7.2 Fitting the test model

How long does it take to fit the test model using methods in Chapter 5? It is interesting to first make some prediction about this.

- Pseudo QR reduction is rich in matrix cross-product $\boldsymbol{X'WX}$, which an optimized BLAS is best at. On an Intel Xeon E5-2650 v2 workstation (see Appendix A for hardware information), OpenBLAS attains a performance of 250 GFLOPs using all 16 CPU cores for matrix-matrix multiplication (revisit §5.2.2 if you have forgotten what "GFLOPs" is). The test model has $n = 1342159$ data and $p = 22571$ coefficients. As is summarized in §4.2.4, pseudo QR reduction involves np^2 FLOP, that is 683763 GFLOP for the test model. Thus I predict that the reduction should takes about 683763 / 250 = 2735 seconds, which is about 45 minutes.

- As is summarized in §4.3.8, the FLOP count for each computation step is different. In particular, many matrix computations of difference performance are involved, so computation time in REML estimation is much more difficult to estimate. A simple yet reliable prediction is to run the REML estimation for just a single Newton-Raphson step, because the total time needed by REML estimation is just this time multiplied by the number of iterations. To start REML estimation, results of pseudo QR reduction, namely $\boldsymbol{X'WX}$ and $\boldsymbol{X'Wy}$ are required. However, there is no need to really do the reduction. For performance measurement, it is sufficient to use a $p \times p$ random matrix for $\boldsymbol{X'WX}$ and a length-p random vector for $\boldsymbol{X'Wy}$. It turns out that for the test model, a single Newton step takes 146 seconds on an Intel Xeon E5-2650 v2 workstation with 16-threaded OpenBLAS. Usually REML estimation takes 10 to 25 steps for convergence, so REML estimation for the test model is unlike to take more than an hour.

Estimation above assured me that fitting the test model is feasible. So I proceeded to actually fit this model. However, practical computation took more time than what I had expected. It turned out that

- REML estimation took 21 steps for convergence, spending 45 minutes, which agrees with my estimate;
- Pseudo QR reduction took 1 hour and 15 minutes, much longer than my estimate;
- Computation of $\hat{\boldsymbol{y}}$ took about 25 minutes.

So, I forgot to take into account of the computation of fitted values, residuals, etc. But, why is $\hat{\boldsymbol{y}}$ computation so expensive? Is it not a matrix-vector multiplication which in principle only involves $2np$ FLOP? On an Intel Xeon E5-2650 v2 workstation matrix-vector multiplication has a performance of 2.9 GFLOPs, so even in serial computing this should just take about 21 seconds. What is contributing to that 25 minutes?

Table 7.1: Termwise number of coefficients p and effective degree of freedom *edf* for fitted test model

	$\hat{f}_0$	$\hat{f}_1$	$\hat{f}_2$	$\hat{f}_3$	$\hat{f}_4$	$\hat{f}_5$	$\hat{f}_6$	$\hat{f}_7$	$\hat{f}_8$	$\hat{f}_9$	$\hat{f}_{10}$
p	5	4	149	2874	39	2308	195	5811	18	2682	702
edf	3.97	1.00	93.38	2679.24	6.79	2214.06	119.86	3646.52	10.06	1413.80	26.22
	$\hat{f}_{11}$	$\hat{f}_{12}$	$\hat{f}_{13}$	$\hat{f}_{14}$	$\hat{f}_{15}$	$\hat{f}_{16}$	$\hat{f}_{17}$	$\hat{f}_{18}$	$\hat{f}_{19}$	$\hat{f}_{20}$	
p	9	9	196	1341	1341	2088	2088	351	351	9	
edf	6.18	2.88	73.65	498.77	513.09	655.69	822.12	215.46	191.36	5.92	

In fact, that $\hat{\boldsymbol{y}}$ computation is time consuming also explains why pseudo QR reduction took longer time than what was predicted. The online updating algorithm for pseudo QR reduction (see Figure 4.6) not just computes matrix cross-product. A chunk of model matrix has to be formed first. Computation of $\hat{\boldsymbol{y}}$ also proceeds by chunks, requiring chunkwise formation of the model matrix. The timing result suggests that this formation costs is somehow rather high. Subtracting 25 minutes from 1 hour 15 minutes gives 50 minutes, i.e., 3000 seconds. This is closer to my estimate of 2735 seconds for matrix cross-product.

From this test model, it is observed that formation of model matrix accounts for nearly 1/3 of the GAM fitting time. For increasingly large n this proportion is even higher. For example, if n is now 10 times as large and p unchanged, both matrix cross-product and model matrix formation will take 500 minutes, but REML estimation will still take 45 minutes, thus the proportion approximates 1/2. This is evidently where the GAM fitting method could / should be improved. I will come back to this issue in the next Chapter.

7.3 Checking the test model

The test model involves 22571 regression coefficients for 1342159 data, ending up with 13201 model degree of freedom. The estimated residual variance is 0.2797, and the variance of logBS data is 1.3945, so the fitted model has an adjusted R-squared of 79.92%. The estimated variance for station-specific random effect and week-specific random effect are respectively 0.0991 and 0.1069. Terwise number of coefficients and effective degree of freedom are summarized in Table 7.1.

Before further checking everything else, I would first check whether there is spatial autocorrelation in residuals. Figure 7.1 presents a set of variograms of spatial residuals from 5 randomly chosen days from every 4 years. It is clear that the model is very inadequate in spatial modelling for the first two decades when the Network was relatively dense.

Figure 7.2 further checks random effects in the model. It turns out that the i.i.d. assumption for station-specific random effect is violated. This is a message that the model has difficulty in spatial modelling.

Figure 7.3 produces the transects plot for the fitted tensor product spline $\hat{f}_7$ that models space-time interaction. The shape of those curve indicates of spatial-temporal extrapolation.

7.4 Summary

I stopped further checking the fitted model, because the message so far is readily clear: the model is very poor. I have to admit that I am unable to develop a reasonably good model. From now on, I will give up model development and focus on improving GAM fitting methods.

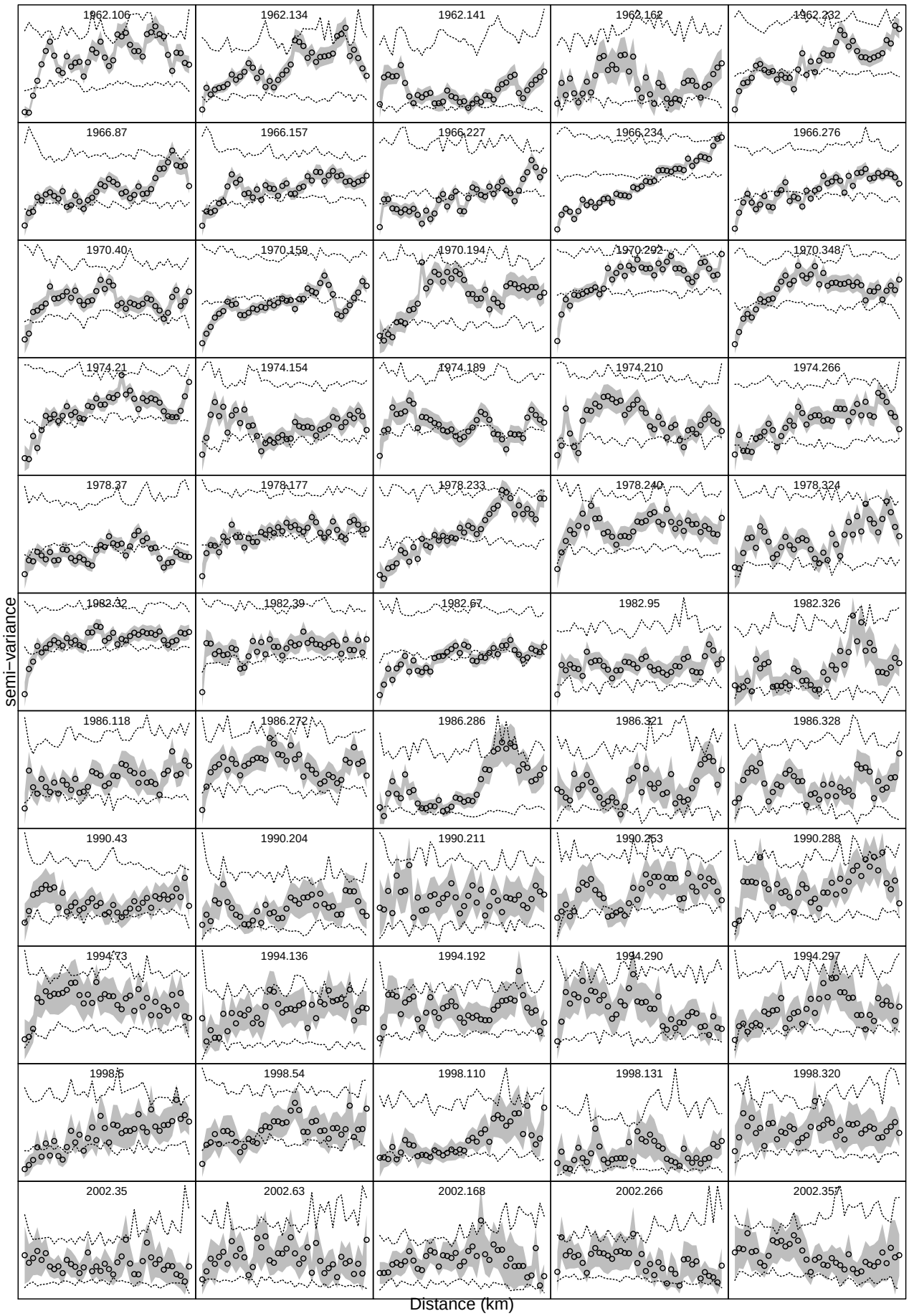


Figure 7.1: For the fitted model 7.1, a set of variograms of spatial residuals from 5 randomly chosen days from every 4 years.

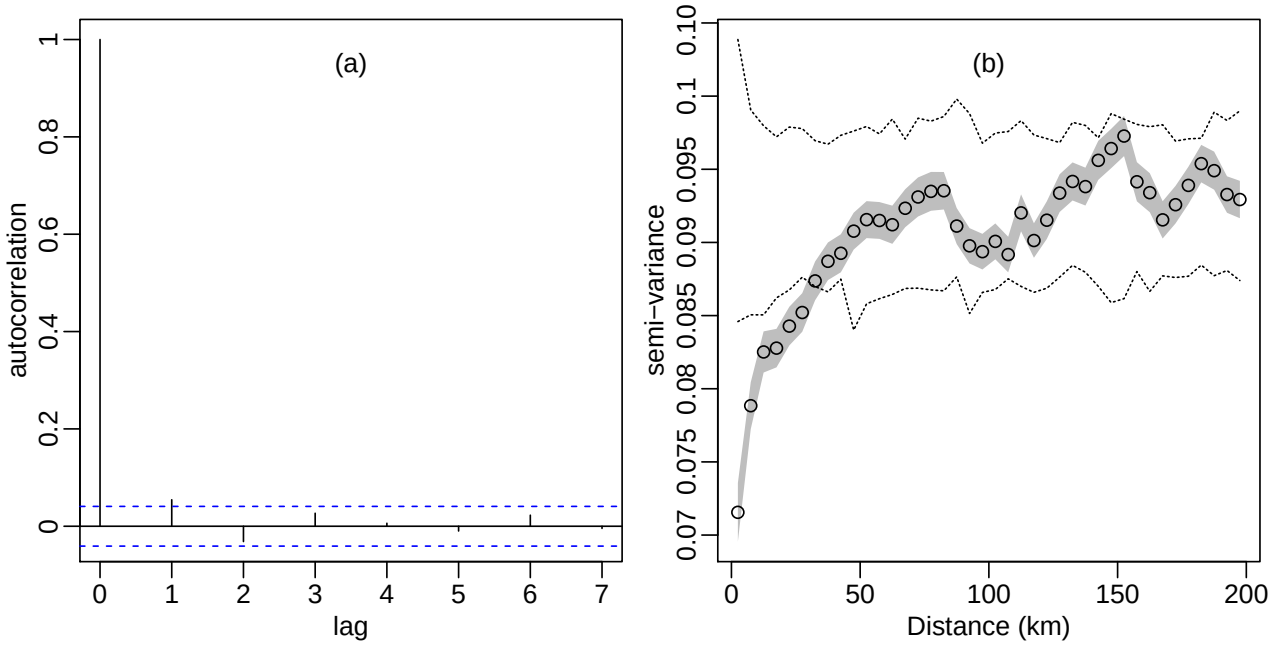


Figure 7.2: Checking i.i.d. random effects in the the fitted model 7.1. Panel (a) is an ACF of week-specific random effects; no autocorrelation is seen. Panel (b) is a variogram of station-specific random effects; however, i.i.d. assumption is violated.

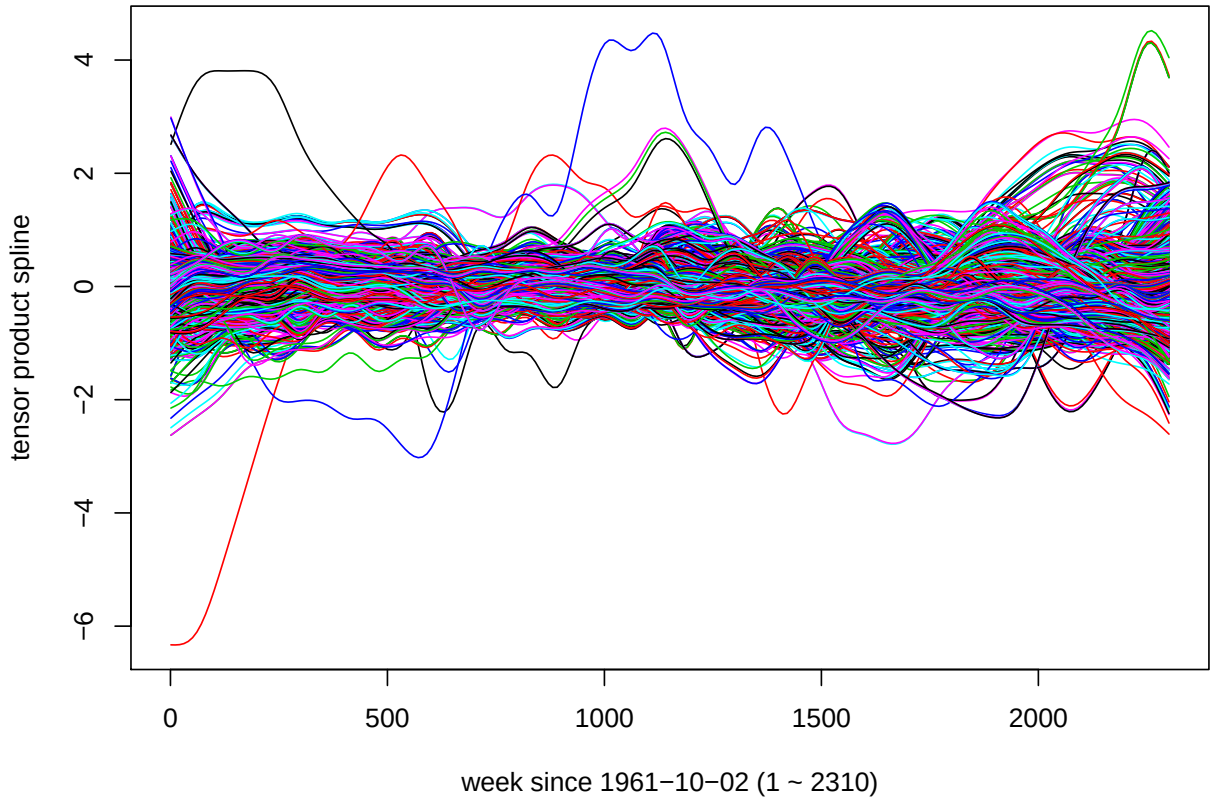


Figure 7.3: Transects plot for the fitted tensor product spline $\hat{f}_7$. These curves are added to the general long term trend for station-specific trend. The shape of those curves indicates there will be unreasonable spatial-temporal extrapolation.

Chapter 8

Discrete pseudo QR reduction

In §7.2 of the previous Chapter, I realized a performance bottleneck for GAM fitting with large datasets: chunkwise formation of $\boldsymbol{X}$ is practically expensive compared with the computation of matrix cross-product $\boldsymbol{X'WX}$ in "online" pseudo QR reduction; it is also what makes $\hat{\boldsymbol{y}}$ computation very time consuming. Thus, faster construction of $\boldsymbol{X}$ is key to faster pseudo QR reduction and $\hat{\boldsymbol{y}}$ computation.

To address this issue, this Chapter will be organized in three stages as follows.

1. §8.1 explains how model matrix construction is conventionally conducted in `gam` function, how it is done in `bam` instead and how their difference eventually leads to the covariate discretization strategy for `bam` that is formally defined in §8.2;
2. §8.3, §8.4 and §8.5 present a set of algorithms for computations of $\boldsymbol{X'WY}$, $\boldsymbol{X'WX}$ and $\hat{\boldsymbol{y}}$ by exploiting packed storage for design matrices after covariate discretization. These algorithms to a large extent eliminate the need for constructing $\boldsymbol{X}$, thus is expected to be significantly faster. Additional attemps to enhance performance of these algorithms via block algorithms / data caching are covered in §8.6.
3. §8.7 experiments the new discrete computation method on the test logBS model 7.1.
4. §8.8 summarizes advantanges and limitations of the new computation method, preluding the next Chapter, the most important Chapter of this thesis.

8.1 Faster model matrix formation for pseudo QR reduction

Suppose we have an additive model

$$\boldsymbol{y} = \boldsymbol{X_0}\boldsymbol{\beta_0} + \sum_{i=1}^{l} \boldsymbol{f_i} + \boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim N(\boldsymbol{0},\ \boldsymbol{W}^{-1}\phi),$$

where in the linear predictor $\boldsymbol{\eta} = \boldsymbol{X_0}\boldsymbol{\beta_0} + \sum_{i=1}^{l} \boldsymbol{f_i}$, there are

- $\boldsymbol{X_0}$, a design matrix from some parametric terms;
- $\boldsymbol{f_1}$, $\boldsymbol{f_2}$, ..., $\boldsymbol{f_l}$, smooth functions represented as splines.

Construction of the model matrix $\boldsymbol{X}$ for this model is to form design matrix $\boldsymbol{X_i}$ for all smooth functions.

Suppose we are to construct the spline basis for a univariate spline (like a cubic regression spline) given a vector of covariate values $\boldsymbol{z} = (z_1,\ z_2,\ \cdots,\ z_n)$. An elementary fact is that if $z_i = z_j$, the i^{th} and j^{th} rows of the design matrix $\boldsymbol{Z}$ are identical. In other words, construction of $\boldsymbol{Z}$ only depends on the set of unique covariate values $\bar{\boldsymbol{z}} = (\bar{z}_1,\ \bar{z}_2,\ \cdots,\ \bar{z}_m)$, where m is the number of unique values. Precisely, we can first construct a *packed design matrix* (with no redundant rows) $\bar{\boldsymbol{Z}}$ from $\bar{\boldsymbol{z}}$, then *unpack* it. To this end, assume we have an index vector $\boldsymbol{k} = (k_1,\ k_2,\ \cdots,\ k_n)$ between $\boldsymbol{z}$ and $\bar{\boldsymbol{z}}$, such that $z_i = \bar{z}_{k_i}$, then the i^{th} row of $\boldsymbol{Z}$ is just the ${k_i}^{\text{th}}$ row of $\bar{\boldsymbol{Z}}$. This construction idea works with other splines, too.

- For a multivariate spline but not a tensor product one, like a bivariate thin-plate regression spline, we have a covariate matrix instead of a covariate vector. The unique covariate values are then the unique rows of this matrix.
- For a tensor product spline, its marginal design matrices can be constructed first with the above method, then after unpacking we can form their row-wise Kronecker product.

The model matrix construction procedure just described has always been implemented in `gam` function of `mgcv`, but not in `bam`. A fundamental assumption of `bam` is that n is very large, then m is also likely large, which is particularly true if some covariate is a continuous variable. Therefore, there may not even be enough RAM to store all packed design matrices. As a result, `bam` constructs $\boldsymbol{X}$ with a different strategy. Let c be a row chunk size (the same one used in "online" model matrix reduction; see §4.2), it has two phases:

1. Prior to "online" model matrix reduction, randomly samples c covariate values from $\boldsymbol{z}$, then appending the minimum and maximum of $\boldsymbol{z}$, construct a design matrix using this subset (which does not necessarily contain all unique values of $\boldsymbol{z}$);
2. During the "online" reduction, construct $\boldsymbol{X}$ chunk by chunk using the initial construction information (like knots locations and transformation matrices) from phase one.

While this keeps memory footprint under control, phase two treats every row of $\boldsymbol{X}$ as if it is unique and computes it, ending up with a complexity proportional to n not m. Thus if $m \ll n$, it is very computationally inefficient.

Many covariates in the logBS dataset only take values from a discrete set. For example, there are only 2874 locations / stations; all time variables are discrete. Other covariates like elevation, temperature and rainfull, while conceptually being continuous variables, are up to an accuracy of 1 metre, 0.1°C and 0.1 millimetre, respectively. They are therefore also discrete with no more than a few hundreds to a few thousands of unique values. In the end, there is $m \ll n$. There will be sufficient RAM and it is more computationally efficient to first construct all packed design matrices, then unpack them during model matrix reduction. A quick experiment showed that for the test model 7.1, the formation time of $\boldsymbol{X}$ was reduced from 25 minutes to just 3 minutes.

In general, many datasets with continuous covariates do not admit $m \ll n$. But is it possible to discretize these covariates, by approximating original covariate values with the nearest values on a regular grid, as if these covariates were measured up to certain accuracy? If this approximation is legitimate in general, the idea of packing and unpacking can be encapsulated into `bam` for efficient model matrix construction.

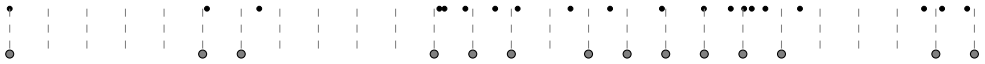


Figure 8.1: An illustration of covariate discretization by rounding. Black dots are original data. Gray dashed lines bin the original data onto a regular grid covering the range of the data. Gray dots are discretized data on the grid. In this example, original data z_i's are 20 samples from Beta(2, 3) distribution, sorted in ascending order. The resolution of the grid is the 60% quantile of $(z_{i+1} - z_i)$.

8.2 Covariate discretization

In this section, I will formulate the idea of covariate discretization brought up at the end of the last section.

For a univariate continuous covariate $\boldsymbol{z} = (z_1,\ z_2,\ \cdots,\ z_n)$, its discretization can be achieved by rounding. See Figure 8.1 for an illustration. In general, define that

- m is the number of unique discretized covariate values, and $\bar{\boldsymbol{z}} = (\bar{z}_1,\ \bar{z}_2,\ \cdots,\ \bar{z}_m)$ is the set of unique values;
- $\boldsymbol{k} = (k_1,\ k_2,\ \cdots,\ k_n)$ is discretization index matching $\boldsymbol{z}$ and $\bar{\boldsymbol{z}}$, such that $z_i \approx \bar{z}_{k_i}$;
- $\bar{\boldsymbol{Z}}$ and $\boldsymbol{Z}$ are respectively the packed design matrix and full design matrix for a spline of $\bar{\boldsymbol{z}}$ and $\boldsymbol{z}$.

For a multivariate continuous covariate, we have a covariate matrix. Discretization is first applied to each of its columns, then unique covariate values are determined by the unique rows of the discretized matrix. Finally, packed design matrix and discretization index can be similarly defined. Under discretization, the i^{th} row of $\boldsymbol{Z}$ can be approximated by the $k_i{}^{\text{th}}$ row of $\bar{\boldsymbol{Z}}$. Inserting some subscript, I will define the collection of $(\bar{\boldsymbol{Z}},\ \boldsymbol{k}_{\bar{z}},\ m_{\bar{z}},\ p_{\bar{z}})$ as the *packed storage* for $\boldsymbol{Z}$, where $p_{\bar{z}}$ is the number of columns of $\bar{\boldsymbol{Z}}$.

Figure 8.2 is a toy example assessing the impact of covariate discretization on estimation. Estimation is still close to truth even if 1000 covariate values are discretized to just 25 unique values.

It should be emphasized that the "covariate discretization", "packed design matrix", "packed storage" and "unpacking" mentioned so far are only related to design matrix $\boldsymbol{X_i}$ of a spline $\boldsymbol{f_i}$. The parametric design matrix $\boldsymbol{X_0}$ is always explicitly constructed as an $n \times p_0$ matrix.

8.3 Computation of $\boldsymbol{X'Wy}$

Consider the linear predictor in §8.1, whose model matrix is $\boldsymbol{X} = (\boldsymbol{X_0}\ \boldsymbol{X_1}\ \boldsymbol{X_2}\ \cdots\ \boldsymbol{X_l})$. Computation of the matrix-vector cross-product $\boldsymbol{X'Wy}$, or $\boldsymbol{X'\tilde{y}}$ where $\tilde{\boldsymbol{y}} = \boldsymbol{Wy}$ is the weighted response vector, may proceed block by block

$$\boldsymbol{X'\tilde{y}} = \begin{pmatrix} \boldsymbol{X_0'\tilde{y}} \\ \boldsymbol{X_1'\tilde{y}} \\ \vdots \\ \boldsymbol{X_l'\tilde{y}} \end{pmatrix}.$$

Given packed storage for $\boldsymbol{X_i}$'s, conventional computation first requires unpacking all $\boldsymbol{X_i}$'s (if $\boldsymbol{X_i}$ is a tensor product design matrix, first unpack all its marginal design matrices, then form their row-wise Kronecker product). In this section, I will describe new computation method for $\boldsymbol{X_i'\tilde{y}}$, where the initial matrix unpacking can be avoided.

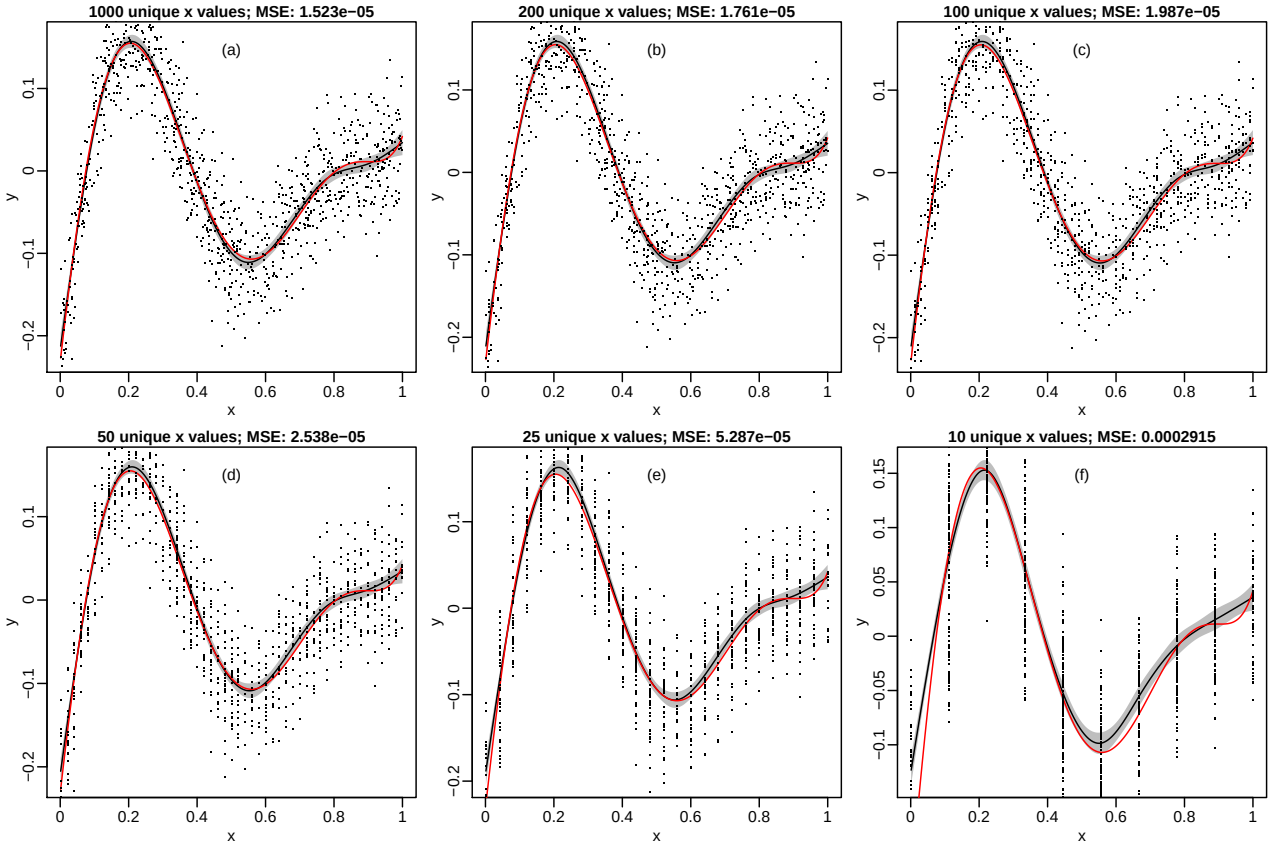


Figure 8.2: A toy example assessing the impact of covariate discretization on estimation. In panel (a), for 1000 uniformly distributed x values on [0, 1], a random polynomial of degree 6 is generated as the true function $y(x)$ (red curve). Gaussian white noise is added with a noise-to-signal ratio of 0.5 (black dots are noise observations). A natural cubic regression spline of 10 knots is fitted (black curve), and the gray ribbon gives 95% pointwise confidence interval of the estimation. In panel (b) to (e), x are discretized to 200, 100, 50, 25 and 10 unique values. The effect of discretization is most obvious from the scatter plot, as they are gradually shrunk toward vertical bars. However, apart from the last panel where 10 unique values are too few, the estimated spline function (black curve) still well approximates the true function (red curve).

For convenience, I will call $\boldsymbol{X_i}$ a *singleton*, if it is a single design matrix, and a *tensor*, if it is a tensor product design matrix with at least two margins. Computation of $\boldsymbol{X_0'\tilde{y}}$ needs no special treatment, as parametric design matrix $\boldsymbol{X_0}$ is not stored in packed format. The focus is to demonstrate algorithms for computing cross-product between a singleton $\boldsymbol{X_i}$ and $\boldsymbol{\tilde{y}}$, and that between a tensor $\boldsymbol{X_i}$ and $\boldsymbol{\tilde{y}}$.

Assume that a singleton $\boldsymbol{X_i}$ has packed storage $(\bar{\boldsymbol{A}},\ \boldsymbol{k}_{\bar{A}},\ m_{\bar{A}},\ p_{\bar{A}})$. An algorithm for $\boldsymbol{X_i'\tilde{y}}$ is as straightforward as Figure 8.3a. For a tensor $\boldsymbol{X_i}$ with s packed margins from $(\bar{\boldsymbol{A}}_{\boldsymbol{1}},\ \boldsymbol{k}_{\bar{A}_1},\ m_{\bar{A}_1},\ p_{\bar{A}_1})$ to $(\bar{\boldsymbol{A}}_{\boldsymbol{s}},\ \boldsymbol{k}_{\bar{A}_s},\ m_{\bar{A}_s},\ p_{\bar{A}_s})$, combine the first $(s-1)$ margins into a single matrix $\dot{\boldsymbol{A}} = \boldsymbol{A_1} \mathbin{\widetilde{\otimes}} \boldsymbol{A_2} \mathbin{\widetilde{\otimes}} \cdots \mathbin{\widetilde{\otimes}} \boldsymbol{A_{s-1}}$, where $\mathbin{\widetilde{\otimes}}$ is the row-wise Kronecker product between two matrices. Then $\boldsymbol{X_i} = \dot{\boldsymbol{A}} \mathbin{\widetilde{\otimes}} \boldsymbol{A_s}$ can be treated as a tensor with only two margins. Applying (1.1) to $\dot{\boldsymbol{A}} \mathbin{\widetilde{\otimes}} \boldsymbol{A_s}$ would give

$$(\dot{\boldsymbol{A}} \mathbin{\widetilde{\otimes}} \boldsymbol{A}_s)'\tilde{\boldsymbol{y}} = \begin{bmatrix} \boldsymbol{A}_s'\mathtt{diag}(\dot{\boldsymbol{A}}(,\ 1)) \\ \boldsymbol{A}_s'\mathtt{diag}(\dot{\boldsymbol{A}}(,\ 2)) \\ \vdots \\ \boldsymbol{A}_s'\mathtt{diag}(\dot{\boldsymbol{A}}(,\ p_{\dot{A}})) \end{bmatrix} \tilde{\boldsymbol{y}} = \begin{bmatrix} \boldsymbol{A}_s'(\mathtt{diag}(\dot{\boldsymbol{A}}(,\ 1))\tilde{\boldsymbol{y}}) \\ \boldsymbol{A}_s'(\mathtt{diag}(\dot{\boldsymbol{A}}(,\ 2))\tilde{\boldsymbol{y}}) \\ \vdots \\ \boldsymbol{A}_s'(\mathtt{diag}(\dot{\boldsymbol{A}}(,\ p_{\dot{A}}))\tilde{\boldsymbol{y}}) \end{bmatrix},$$

where $p_{\dot{A}} = \prod_{j=1}^{s-1} p_{\bar{A}_j}$. This implies that $(\dot{\boldsymbol{A}} \mathbin{\widetilde{\otimes}} \boldsymbol{A}_s)'\tilde{\boldsymbol{y}}$ can be obtained by computing $\boldsymbol{A}_s'\tilde{\boldsymbol{y}}_{\boldsymbol{k}}$ block by block, where $\tilde{\boldsymbol{y}}_{\boldsymbol{k}} = \mathtt{diag}(\dot{\boldsymbol{A}}(,\ k))\tilde{\boldsymbol{y}}$. Figure 8.3b demonstrates the algorithm.

Implementation of algorithm 8.3b requires extracting the k^{th} column from the tensor product design matrix $\dot{\boldsymbol{A}}$. It can be shown that this column is an Hadamard product[1]

$$\dot{\boldsymbol{A}}(,\ k) = \boldsymbol{A_1}(,\ k_1) \circ \boldsymbol{A_2}(,\ k_2) \circ \cdots \circ \boldsymbol{A_{s-1}}(,\ k_{s-1}),$$

[1]For two vectors $\boldsymbol{x} = \begin{pmatrix} x_1 & x_2 & \cdots & x_n \end{pmatrix}$ and $\boldsymbol{y} = \begin{pmatrix} y_1 & y_2 & \cdots & y_n \end{pmatrix}$, their Hadamard product is the element-wise

for some $k_1,\ k_2, \cdots,\ k_{s-1}$. Here, using zero-based index (that is, if a matrix has p columns, they are columns $0,\ 1,\ \cdots,\ (p-1)$) turns out convenient. The k_j index can be determined most easily in a two-margin scenario $\dot{\boldsymbol{A}}(,\ k) = \boldsymbol{A_1}(,\ k_1) \circ \boldsymbol{A_2}(,\ k_2)$. By applying (1.1) to $\boldsymbol{A_1} \mathbin{\widetilde{\otimes}} \boldsymbol{A_2}$, it can be derived that $k_2 = [k/p_{\boldsymbol{A_2}}],\ k_1 = \lfloor k/p_{\boldsymbol{A_2}} \rfloor$, where $[a/b]$ is the *modulo* function and $\lfloor a/b \rfloor$ is the *floor* function. For more than two margins, say three margins, a recursion applies. Collect the first two margins into a single matrix. Then if $\dot{\boldsymbol{A}}(,\ k) = (\boldsymbol{A_1} \mathbin{\widetilde{\otimes}} \boldsymbol{A_2})(,\ \tilde{k}) \circ \boldsymbol{A_3}(,\ k_3)$, the previous two-margin result gives $k_3 = [k/p_{\boldsymbol{A_3}}]$ and $\tilde{k} = \lfloor k/p_{\boldsymbol{A_3}} \rfloor$. Now applying the same to $(\boldsymbol{A_1} \mathbin{\widetilde{\otimes}} \boldsymbol{A_2})(,\ \tilde{k}) = \boldsymbol{A_1}(,\ k_1) \circ \boldsymbol{A_2}(,\ k_2)$ gives $k_2 = [\tilde{k}/p_{\boldsymbol{A_2}}]$ and $k_1 = \lfloor \tilde{k}/p_{\boldsymbol{A_2}} \rfloor$. In general, the k^{th} column can be computed using algorithm in Figure 8.3c.

Algorithms in Figures 8.3a and 8.3b respectively involve $O(n)+O(m_{\bar{A}}p_{\bar{A}})$ and $O(snp_{\dot{A}})+O(m_{\bar{A}_s}p_{\dot{A}}p_{\bar{A}_s})$ FLOP, smaller than the $O(np_{\bar{A}})$ and $O(np_{\dot{A}}p_{\bar{A}_s})$ FLOP via conventional computation method with initial matrix unpacking. These algorithms are expected to be $O(\min\{p_{\bar{A}},\ n/m_{\bar{A}}\})$ times faster for a singleton $\boldsymbol{X_i}$, and $O(\min\{p_{\bar{A}_s},\ n/m_{\bar{A}_s}\})$ times faster for a tensor $\boldsymbol{X_i}$. The latter result implies that computations of $(\boldsymbol{A_1} \mathbin{\widetilde{\otimes}} \boldsymbol{A_2})'\tilde{\boldsymbol{y}}$ and $(\boldsymbol{A_2} \mathbin{\widetilde{\otimes}} \boldsymbol{A_1})'\tilde{\boldsymbol{y}}$ are not equally fast if $p_{\bar{A}_1} \neq p_{\bar{A}_2}$. Generally for a tensor $\boldsymbol{X_i}$, if its largest margin (the one with the maximal $p_{\bar{A}_j}$) is the last margin, the maximal speedup can be attained. Therefore in `mgcv` implementation, it is worth exchanging the largest margin to the last margin during construction of a tensor product spline.

The centring constraint described in §4.1.6 implies that in practice, `mgcv` would estimate $\tilde{\boldsymbol{\beta}}_{\boldsymbol{i}}$ not $\boldsymbol{\beta_i}$, so the cross-product that is really needed is $(\boldsymbol{X_i Q_i}^{\perp})'\tilde{\boldsymbol{y}}$. It turns out that we can first work out $\boldsymbol{z} = \boldsymbol{X_i}'\tilde{\boldsymbol{y}}$ using previously described algorithms, then compute $(\boldsymbol{Q_i}^{\perp})'\boldsymbol{z}$ as a post-processing.

8.4 Computation of $\boldsymbol{X'WX}$

Consider the linear predictor in §8.1, whose model matrix is $\boldsymbol{X} = (\boldsymbol{X_0}\ \boldsymbol{X_1}\ \boldsymbol{X_2}\ \cdots\ \boldsymbol{X_l})$. Computation of the matrix cross-product $\boldsymbol{X'WX}$ may proceed block by block

$$\boldsymbol{X'WX} = \begin{pmatrix} \boldsymbol{X_0'WX_0} & \boldsymbol{X_0'WX_1} & \cdots & \boldsymbol{X_0'WX_l} \\ \boldsymbol{X_1'WX_0} & \boldsymbol{X_1'WX_1} & \cdots & \boldsymbol{X_1'WX_l} \\ \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{X_l'WX_0} & \boldsymbol{X_l'WX_1} & \cdots & \boldsymbol{X_l'WX_l} \end{pmatrix},$$

where only the lower triangular part needs be computed due to symmetry. Given packed storage for $\boldsymbol{X_i}$'s, conventional computation requires unpacking all $\boldsymbol{X_i}$'s (if $\boldsymbol{X_i}$ is a tensor product design matrix, first unpack all its marginal design matrices, then form their row-wise Kronecker product) before computing $\boldsymbol{X_i'WX_j}$. However, using algorithms in §8.3, computation of $\boldsymbol{X_i'WX_j}$ can be faster. Consider only unpacking $\boldsymbol{X_j}$ into a full design matrix $\boldsymbol{Y}$, then $\boldsymbol{X_i'WX_j} = \boldsymbol{X_i'WY}$ can be computed using algorithms in Figures 8.4a and 8.4b, which are straightforward generalization of algorithms in Figures 8.3a and 8.3b. Compared with conventional computation where both $\boldsymbol{X_i}$ and $\boldsymbol{X_j}$ are unpacked, these algorithms do not require unpacking $\boldsymbol{X_i}$, and are expected to be $O(\min\{p_{\bar{A}},\ n/m_{\bar{A}}\})$ times faster for a singleton $\boldsymbol{X_i}$, and $O(\min\{p_{\bar{A}_s},\ n/m_{\bar{A}_s}\})$ times faster for a tensor $\boldsymbol{X_i}$.

In general, fully unpacking $\boldsymbol{X_j}$ would require too much RAM if n is large. A workaround is to only unpack a column chunk at one time, as implemented by algorithms in Figure 8.5a and 8.5b. With a chunk size b, the memory footprint is bounded by $O(nb)$. Note that for a tensor $\boldsymbol{X_i}$, the value of b would affect the complexity of the algorithm[2]. As a result, the algorithm is $O(\min\{p_{\bar{A}_s}/(1 + s/b),\ n/m_{\bar{A}_s}\})$ times faster than conventional computation. To maximize this speedup factor, s/b should be as small as possible. A reasonable choice is $b = 20s$ so that $s/b = 0.05$.

product $\boldsymbol{x} \circ \boldsymbol{y} = \begin{pmatrix} x_1y_1 & x_2y_2 & \cdots & x_ny_n \end{pmatrix}$. Hadamard product is commutative and associative, and it is straightforward to extend to more than two vectors.

[2]Because after chunking, every column of $\dot{\boldsymbol{A}}$ is respectively extracted $(p_{\bar{B}}/b)$ and (p_B/b) times rather than only once for a singleton $\boldsymbol{X_j}$ and a tensor $\boldsymbol{X_j}$.

initialize a length-$m_{\bar{A}}$ vector of zeros
$\bar{\boldsymbol{y}} = \text{zeros}(m_{\bar{\boldsymbol{A}}})$
vector aggregation, $O(n)$
for $i = 1 : n$
  $\bar{\boldsymbol{y}}(\boldsymbol{k}_{\bar{A}}(i)) \mathrel{+}= \tilde{\boldsymbol{y}}(i)$
matrix-vector cross-product, $O(m_{\bar{A}} p_{\bar{A}})$
$\boldsymbol{A}'\tilde{\boldsymbol{y}} = \bar{\boldsymbol{A}}'\bar{\boldsymbol{y}}$

(a) $\boldsymbol{X}_{\boldsymbol{i}}'\tilde{\boldsymbol{y}} = \boldsymbol{A}'\tilde{\boldsymbol{y}}$, where $\boldsymbol{X}_{\boldsymbol{i}}$ is a singleton with packed storage $(\bar{\boldsymbol{A}},\ \boldsymbol{k}_{\bar{A}},\ m_{\bar{A}},\ p_{\bar{A}})$. FLOP count for each computation step is given in gray text, and the algorithm involves $O(n) + O(m_{\bar{A}} p_{\bar{A}})$ FLOP.

$p_{\dot{A}} = \prod_{j=1}^{s-1} p_{\bar{A}_j}$
for $k = 1 : p_{\dot{A}}$
  compute $\tilde{\boldsymbol{y}}_{\boldsymbol{k}}$, O(sn)
  compute $\tilde{\boldsymbol{y}}_{\boldsymbol{k}} = \texttt{diag}(\dot{\boldsymbol{A}}(,\ k))\tilde{\boldsymbol{y}}$ using algorithm in Figure 8.3c
  compute elements $(kp_{\bar{A}_s} - p_{\bar{A}_s} + 1)$ to $kp_{\bar{A}_s}$ of $\boldsymbol{X}_{\boldsymbol{i}}'\tilde{\boldsymbol{y}}$, $O(n) + O(m_{\bar{A}_s} p_{\bar{A}_s})$
  compute $\boldsymbol{z} = \boldsymbol{A}_{\boldsymbol{s}}'\tilde{\boldsymbol{y}}_{\boldsymbol{k}}$ using algorithm in Figure 8.3a
  $(\boldsymbol{X}_{\boldsymbol{i}}'\tilde{\boldsymbol{y}})((kp_{\bar{A}_s} - p_{\bar{A}_s} + 1) : kp_{\bar{A}_s}) = \boldsymbol{z}$

(b) $\boldsymbol{X}_{\boldsymbol{i}}'\tilde{\boldsymbol{y}} = (\dot{\boldsymbol{A}} \mathbin{\widetilde{\otimes}} \boldsymbol{A}_{\boldsymbol{s}})'\tilde{\boldsymbol{y}}$, where $\boldsymbol{X}_{\boldsymbol{i}}$ is a tensor with s packed margins $(\bar{\boldsymbol{A}}_{\boldsymbol{j}},\ \boldsymbol{k}_{\bar{A}_j},\ m_{\bar{A}_j},\ p_{\bar{A}_j})$ and $\dot{\boldsymbol{A}} = \boldsymbol{A}_{\boldsymbol{1}} \mathbin{\widetilde{\otimes}} \boldsymbol{A}_{\boldsymbol{2}} \mathbin{\widetilde{\otimes}} \cdots \mathbin{\widetilde{\otimes}} \boldsymbol{A}_{\boldsymbol{s-1}}$. FLOP count for each computation step is given in gray text, and the algorithm involves $O(snp_{\dot{A}}) + O(m_{\bar{A}_s} p_{\dot{A}} p_{\bar{A}_s})$ FLOP.

initialize a length-n vector of ones, $O(n)$
$\boldsymbol{a} = \text{ones}(n)$
initialize $\tilde{k}$
$\tilde{k} = k$
for $j = (s-1) : 1$
  compute k_j
  $k_j = [\tilde{k}/p_{\bar{A}_j}]$
  update $\tilde{k}$
  $\tilde{k} = \lfloor \tilde{k}/p_{\bar{A}_j} \rfloor$
  compute Hadamard product, $O(n)$
  $\boldsymbol{a} = \boldsymbol{a} \circ \boldsymbol{A}_{\boldsymbol{j}}(,\ k_j)$
store result
$\dot{\boldsymbol{A}}(,\ k) = \boldsymbol{a}$

(c) Compute the k^{th} column of $\dot{\boldsymbol{A}} = \boldsymbol{A}_{\boldsymbol{1}} \mathbin{\widetilde{\otimes}} \boldsymbol{A}_{\boldsymbol{2}} \mathbin{\widetilde{\otimes}} \cdots \mathbin{\widetilde{\otimes}} \boldsymbol{A}_{\boldsymbol{s-1}}$. Here, zero-based index is used for convenience, that is, if a matrix has p columns, they are columns $0,\ 1,\ \cdots,\ (p-1)$. When $\boldsymbol{A}_j$ is stored in packed format $(\bar{\boldsymbol{A}}_{\boldsymbol{j}},\ \boldsymbol{k}_{\bar{A}_j},\ m_{\bar{A}_j},\ p_{\bar{A}_j})$, the Hadamard product can be computed as $\boldsymbol{a} = \boldsymbol{a} \circ \bar{\boldsymbol{A}}_{\boldsymbol{j}}(\boldsymbol{k}_{\bar{A}_j},\ k_j)$. Note that this algorithm essentially computes $\dot{\boldsymbol{A}}(,\ k) \circ \boldsymbol{a}$, or equivalently $\texttt{diag}(\dot{\boldsymbol{A}}(,\ k))\boldsymbol{a}$, but it returns $\dot{\boldsymbol{A}}(,\ k)$ if $\boldsymbol{a}$ is initially a vector of ones. The algorithm involves $O(sn)$ FLOP.

Figure 8.3: Computation of $\boldsymbol{X}_{\boldsymbol{i}}'\boldsymbol{W}\boldsymbol{y} = \boldsymbol{X}_{\boldsymbol{i}}'\tilde{\boldsymbol{y}}$, where $\boldsymbol{X}_{\boldsymbol{i}}$ is a singleton or tensor, and $\tilde{\boldsymbol{y}} = \boldsymbol{W}\boldsymbol{y}$ is the weighted response vector.

initialize a $m_{\bar{A}} \times p_Y$ matrix of zeros
$\bar{\boldsymbol{Y}} = \text{zeros}(m_{\bar{\boldsymbol{A}}},\ p_{\boldsymbol{Y}})$
matrix row-aggregation, $O(np_Y)$
for $i = 1 : n$
  $\bar{\boldsymbol{Y}}(\boldsymbol{k}_{\bar{\boldsymbol{A}}}(i),\ ) \mathrel{+}= \tilde{\boldsymbol{Y}}(i,\ )$
matrix cross-product, $O(m_{\bar{A}} p_{\bar{A}} p_Y)$
$\boldsymbol{A}'\tilde{\boldsymbol{Y}} = \bar{\boldsymbol{A}}'\bar{\boldsymbol{Y}}$

(a) $\boldsymbol{X}_{\boldsymbol{i}}'\tilde{\boldsymbol{Y}} = \boldsymbol{A}'\tilde{\boldsymbol{Y}}$, where $\boldsymbol{X}_{\boldsymbol{i}}$ is a singleton with packed storage $(\bar{\boldsymbol{A}},\ \boldsymbol{k}_{\bar{\boldsymbol{A}}},\ m_{\bar{\boldsymbol{A}}},\ p_{\bar{\boldsymbol{A}}})$. FLOP count for each computation step is given in gray text, and the algorithm involves $O(np_{\boldsymbol{Y}}) + O(m_{\bar{\boldsymbol{A}}} p_{\bar{\boldsymbol{A}}} p_{\boldsymbol{Y}})$ FLOP.

$p_{\dot{\boldsymbol{A}}} = \prod_{j=1}^{s-1} p_{\bar{\boldsymbol{A}}_j}$
for $k = 1 : p_{\dot{\boldsymbol{A}}}$
  form k^{th} column of $\dot{\boldsymbol{A}} = \boldsymbol{A_1} \mathbin{\widetilde{\otimes}} \boldsymbol{A_2} \mathbin{\widetilde{\otimes}} \cdots \mathbin{\widetilde{\otimes}} \boldsymbol{A_{s-1}}$, $O(sn)$
  compute $\boldsymbol{a} = \dot{\boldsymbol{A}}(,\ k)$ using algorithm in Figure 8.3c
  matrix row-scaling, $O(np_{\boldsymbol{Y}})$
  $\tilde{\boldsymbol{Y}}_{\boldsymbol{k}} = \texttt{diag}(\boldsymbol{a})\tilde{\boldsymbol{Y}}$
  compute rows $(kp_{\bar{\boldsymbol{A}}_s} - p_{\bar{\boldsymbol{A}}_s} + 1)$ to $kp_{\bar{\boldsymbol{A}}_s}$ of $\boldsymbol{X}_{\boldsymbol{i}}'\tilde{\boldsymbol{Y}}$, $O(np_{\boldsymbol{Y}}) + O(m_{\bar{\boldsymbol{A}}_s} p_{\bar{\boldsymbol{A}}_s} p_{\boldsymbol{Y}})$
  compute $\boldsymbol{Z} = \boldsymbol{A}_{\boldsymbol{s}}'\tilde{\boldsymbol{Y}}_{\boldsymbol{k}}$ using algorithm in Figure 8.4a
  $(\boldsymbol{X}_{\boldsymbol{i}}'\tilde{\boldsymbol{Y}})((kp_{\bar{\boldsymbol{A}}_s} - p_{\bar{\boldsymbol{A}}_s} + 1) : kp_{\bar{\boldsymbol{A}}_s},\ ) = \boldsymbol{Z}$

(b) $\boldsymbol{X}_{\boldsymbol{i}}'\tilde{\boldsymbol{Y}} = (\dot{\boldsymbol{A}} \mathbin{\widetilde{\otimes}} \boldsymbol{A}_{\boldsymbol{s}})'\tilde{\boldsymbol{Y}}$, where $\boldsymbol{X}_{\boldsymbol{i}}$ is a tensor with s packed margins from $(\bar{\boldsymbol{A}}_{\boldsymbol{1}},\ \boldsymbol{k}_{\bar{\boldsymbol{A}}_1},\ m_{\bar{\boldsymbol{A}}_1},\ p_{\bar{\boldsymbol{A}}_1})$ to $(\bar{\boldsymbol{A}}_{\boldsymbol{s}},\ \boldsymbol{k}_{\bar{\boldsymbol{A}}_s},\ m_{\bar{\boldsymbol{A}}_s},\ p_{\bar{\boldsymbol{A}}_s})$. FLOP count for each computation step is given in gray text, and the algorithm involves $O(snp_{\dot{\boldsymbol{A}}}) + O(np_{\dot{\boldsymbol{A}}} p_{\boldsymbol{Y}}) + O(m_{\bar{\boldsymbol{A}}_s} p_{\dot{\boldsymbol{A}}} p_{\bar{\boldsymbol{A}}_s} p_{\boldsymbol{Y}})$ FLOP.

Figure 8.4: Computation of $\boldsymbol{X}_{\boldsymbol{i}}'\boldsymbol{W}\boldsymbol{Y} = \boldsymbol{X}_{\boldsymbol{i}}'\tilde{\boldsymbol{Y}}$, where $\boldsymbol{X}_{\boldsymbol{i}}$ is a singleton or tensor and $\tilde{\boldsymbol{Y}} = \boldsymbol{W}\boldsymbol{Y}$ has $p_{\boldsymbol{Y}}$ columns.

Note that computations of $\boldsymbol{X}_{\boldsymbol{i}}'\boldsymbol{W}\boldsymbol{X}_{\boldsymbol{j}}$ and $\boldsymbol{X}_{\boldsymbol{j}}'\boldsymbol{W}\boldsymbol{X}_{\boldsymbol{i}}$ are not equally fast using algorithms above. For example, assuming that $\boldsymbol{X}_{\boldsymbol{i}}$ and $\boldsymbol{X}_{\boldsymbol{j}}$ are both tensors, the former yields $O(p_{\bar{\boldsymbol{A}}_s})$ speedup, while the latter yields $O(p_{\bar{\boldsymbol{B}}_t})$ speedup. This imples that if $p_{\bar{\boldsymbol{A}}_s} < p_{\bar{\boldsymbol{B}}_t}$, it is more efficient to compute $\boldsymbol{X}_{\boldsymbol{j}}'\boldsymbol{W}\boldsymbol{X}_{\boldsymbol{i}}$, then transpose it to get $\boldsymbol{X}_{\boldsymbol{i}}'\boldsymbol{W}\boldsymbol{X}_{\boldsymbol{j}}$. In `mgcv`, a run-time decision can be made on this.

With centring constraint, the cross-product needed is $(\boldsymbol{X}_{\boldsymbol{i}}\boldsymbol{Q}_{\boldsymbol{i}}^{\perp})'\boldsymbol{W}\boldsymbol{X}_{\boldsymbol{j}}\boldsymbol{Q}_{\boldsymbol{j}}^{\perp}$. We can first get $\boldsymbol{Z} = \boldsymbol{X}_{\boldsymbol{i}}'\boldsymbol{W}\boldsymbol{X}_{\boldsymbol{j}}$ using previously described algorithms, then compute $(\boldsymbol{Q}_{\boldsymbol{i}}^{\perp})'\boldsymbol{Z}\boldsymbol{Q}_{\boldsymbol{j}}^{\perp}$ as a post-processing.

8.5 Computation of $\hat{\boldsymbol{y}}$

Now let us consider the computation of $\hat{\boldsymbol{y}}$. For the linear prediction in §8.1, there is $\hat{\boldsymbol{y}} = \sum_{i=0}^{l} \hat{\boldsymbol{y}}_{\boldsymbol{i}} = \sum_{i=0}^{l} \boldsymbol{X}_{\boldsymbol{i}}\hat{\boldsymbol{\beta}}_{\boldsymbol{i}}$. Conventional computation would first unpack all $\boldsymbol{X}_{\boldsymbol{i}}$'s from their packed storage then do matrix-vector multiplication. But a more efficient method is possible without matrix unpacking. To simplify notations in the following, let $\boldsymbol{b}_{\boldsymbol{i}} = \hat{\boldsymbol{\beta}}_{\boldsymbol{i}}$.

If $\boldsymbol{X}_{\boldsymbol{i}}$ is a singleton, then $\hat{\boldsymbol{y}}_{\boldsymbol{i}}$ can be straightforward obtained with the algorithm in Figure 8.6a which is $O(\min\{p_{\bar{\boldsymbol{A}}},\ n/m_{\bar{\boldsymbol{A}}}\})$ times faster than conventional computation. If $\boldsymbol{X}_{\boldsymbol{i}}$ is a tensor with s margins from $(\bar{\boldsymbol{A}}_{\boldsymbol{1}},\ \boldsymbol{k}_{\bar{\boldsymbol{A}}_1},\ m_{\bar{\boldsymbol{A}}_1},\ p_{\bar{\boldsymbol{A}}_1})$ to $(\bar{\boldsymbol{A}}_{\boldsymbol{s}},\ \boldsymbol{k}_{\bar{\boldsymbol{A}}_s},\ m_{\bar{\boldsymbol{A}}_s},\ p_{\bar{\boldsymbol{A}}_s})$, there is

$$\hat{\boldsymbol{y}}_{\boldsymbol{i}} = (\dot{\boldsymbol{A}} \mathbin{\widetilde{\otimes}} \boldsymbol{A}_{\boldsymbol{s}})\boldsymbol{b}_{\boldsymbol{i}} = \begin{pmatrix} \cdots & \texttt{diag}(\dot{\boldsymbol{A}}(,\ k))\boldsymbol{A}_{\boldsymbol{s}} & \cdots \end{pmatrix} \begin{pmatrix} \vdots \\ \boldsymbol{b}_{\boldsymbol{i}}((kp_{\bar{\boldsymbol{A}}_s} - p_{\bar{\boldsymbol{A}}_s} + 1) : kp_{\bar{\boldsymbol{A}}_s}) \\ \vdots \end{pmatrix}$$

$$= \sum_{k=1}^{p_{\dot{\boldsymbol{A}}}} \texttt{diag}(\dot{\boldsymbol{A}}(,\ k))\boldsymbol{A}_{\boldsymbol{s}}\boldsymbol{b}_{\boldsymbol{i}}((kp_{\bar{\boldsymbol{A}}_s} - p_{\bar{\boldsymbol{A}}_s} + 1) : kp_{\bar{\boldsymbol{A}}_s})$$

$k = 1$
while $(k \leq p_{\boldsymbol{B}})$
  determine column chunk size
  $\tilde{b} = \min\{b,\ p_{\boldsymbol{B}} - k + 1\}$
  unpack columns k to $(k + \tilde{b} - 1)$ of $\boldsymbol{B}$, $O(n\tilde{b})$
  for $\tilde{k} = 1 : \tilde{b}$
    $\boldsymbol{Y}(,\ \tilde{k}) = \bar{\boldsymbol{B}}(\boldsymbol{k}_{\bar{B}},\ k + \tilde{k} - 1)$
  matrix weighting ($\boldsymbol{Y}$ can be overwritten), $O(n\tilde{b})$
  $\boldsymbol{Y} = \boldsymbol{W}\boldsymbol{Y}$
  compute columns k to $(k + \tilde{b} - 1)$ of $\boldsymbol{X}_{\boldsymbol{i}}'\boldsymbol{W}\boldsymbol{X}_{\boldsymbol{j}}$
  $O(n\tilde{b}) + O(m_{\bar{A}}p_{\bar{A}}\tilde{b})$ or $O(snp_{\dot{A}}) + O(np_{\dot{A}}\tilde{b}) + O(m_{\bar{A}_s}p_{\dot{A}}p_{\bar{A}_s}\tilde{b})$
  compute $\boldsymbol{Z} = \boldsymbol{X}_{\boldsymbol{i}}'\boldsymbol{Y}$ using algorithm in Figure 8.4a or 8.4b
  $(\boldsymbol{X}_{\boldsymbol{i}}'\boldsymbol{W}\boldsymbol{X}_{\boldsymbol{j}})(,\ k : (k + \tilde{b} - 1)) = \boldsymbol{Z}$
  proceed to column chunk
  $k = k + \tilde{b}$

(a) $\boldsymbol{X}_{\boldsymbol{i}}'\boldsymbol{W}\boldsymbol{X}_{\boldsymbol{j}} = \boldsymbol{X}_{\boldsymbol{i}}'\boldsymbol{W}\boldsymbol{B}$, where $\boldsymbol{X}_{\boldsymbol{j}}$ is a singleton with packed storage $(\bar{\boldsymbol{B}},\ \boldsymbol{k}_{\bar{B}},\ m_{\bar{B}},\ p_{\bar{B}})$. FLOP count for each computation step is given in gray text, and the algorithm involves $O(np_{\bar{B}}) + O(m_{\bar{A}}p_{\bar{A}}p_{\bar{B}})$ FLOP for a singleton $\boldsymbol{X}_{\boldsymbol{i}}$ and $O(np_{\bar{B}}) + O(snp_{\dot{A}}p_{\bar{B}}/b) + O(np_{\dot{A}}p_{\bar{B}}) + O(m_{\bar{A}_s}p_{\dot{A}}p_{\bar{A}_s}p_{\bar{B}})$ FLOP for a tensor $\boldsymbol{X}_{\boldsymbol{i}}$.

$p_{\boldsymbol{B}} = \prod_{j=1}^{t} p_{\bar{B}_j}$
$k = 1$
while $(k \leq p_{\boldsymbol{B}})$
  determine column chunk size
  $\tilde{b} = \min\{b,\ p_{\boldsymbol{B}} - k + 1\}$
  unpack columns k to $(k + \tilde{b} - 1)$ of $\boldsymbol{B_1} \widetilde{\otimes} \boldsymbol{B_2} \widetilde{\otimes} \cdots \widetilde{\otimes} \boldsymbol{B_t}$, $O(nt\tilde{b})$
  for $\tilde{k} = 1 : \tilde{b}$
    $\boldsymbol{Y}(,\ \tilde{k}) = (\boldsymbol{B_1} \widetilde{\otimes} \boldsymbol{B_2} \widetilde{\otimes} \cdots \widetilde{\otimes} \boldsymbol{B_t})(,\ k + \tilde{k} - 1)$
  matrix weighting ($\boldsymbol{Y}$ can be overwritten), $O(n\tilde{b})$
  $\boldsymbol{Y} = \boldsymbol{W}\boldsymbol{Y}$
  compute columns k to $(k + \tilde{b} - 1)$ of $\boldsymbol{X}_{\boldsymbol{i}}'\boldsymbol{W}\boldsymbol{X}_{\boldsymbol{j}}$
  $O(n\tilde{b}) + O(m_{\bar{A}}p_{\bar{A}}\tilde{b})$ or $O(snp_{\dot{A}}) + O(np_{\dot{A}}\tilde{b}) + O(m_{\bar{A}_s}p_{\dot{A}}p_{\bar{A}_s}\tilde{b})$
  compute $\boldsymbol{Z} = \boldsymbol{X}_{\boldsymbol{i}}'\boldsymbol{Y}$ using algorithm in Figure 8.4a or 8.4b
  $(\boldsymbol{X}_{\boldsymbol{i}}'\boldsymbol{W}\boldsymbol{X}_{\boldsymbol{j}})(,\ k : (k + \tilde{b} - 1)) = \boldsymbol{Z}$
  proceed to column chunk
  $k = k + \tilde{b}$

(b) $\boldsymbol{X}_{\boldsymbol{i}}'\boldsymbol{W}\boldsymbol{X}_{\boldsymbol{j}} = \boldsymbol{X}_{\boldsymbol{i}}'\boldsymbol{W}(\boldsymbol{B_1} \widetilde{\otimes} \boldsymbol{B_2} \widetilde{\otimes} \cdots \widetilde{\otimes} \boldsymbol{B_t})$, where $\boldsymbol{X}_{\boldsymbol{j}}$ is a tensor with t packed margins from $(\bar{\boldsymbol{B}}_{\boldsymbol{1}},\ \boldsymbol{k}_{\bar{B}_1},\ m_{\bar{B}_1},\ p_{\bar{B}_1})$ to $(\bar{\boldsymbol{B}}_{\boldsymbol{t}},\ \boldsymbol{k}_{\bar{B}_t},\ m_{\bar{B}_t},\ p_{\bar{B}_t})$. FLOP count for each computation step is given in gray text, and the algorithm involves $O(ntp_{\boldsymbol{B}}) + O(m_{\bar{A}}p_{\bar{A}}p_{\boldsymbol{B}})$ FLOP for a singleton $\boldsymbol{X}_{\boldsymbol{i}}$ and $O(ntp_{\boldsymbol{B}}) + O(snp_{\dot{A}}p_{\boldsymbol{B}}/b) + O(np_{\dot{A}}p_{\boldsymbol{B}}) + O(m_{\bar{A}_s}p_{\dot{A}}p_{\bar{A}_s}p_{\boldsymbol{B}})$ FLOP for a tensor $\boldsymbol{X}_{\boldsymbol{i}}$.

Figure 8.5: Computation of $\boldsymbol{X}_{\boldsymbol{i}}'\boldsymbol{W}\boldsymbol{X}_{\boldsymbol{j}}$, where $\boldsymbol{X}_{\boldsymbol{i}}$ and $\boldsymbol{X}_{\boldsymbol{j}}$ can be singletons or tensors. (If $\boldsymbol{X}_{\boldsymbol{j}}$ is the parametric design matrix $\boldsymbol{X}_{\boldsymbol{0}}$, computation of $\boldsymbol{X}_{\boldsymbol{i}}'\boldsymbol{W}\boldsymbol{X}_{\boldsymbol{0}}$ is straightforward using algorithms in Figure 8.4a or 8.4b by setting $\boldsymbol{Y} = \boldsymbol{X}_{\boldsymbol{0}}$.)

initialize a length-n vector of zeros, O(n)
$\hat{\boldsymbol{y}}_{\boldsymbol{i}} = \text{zeros}(n)$
matrix-vector multiplication, $O(m_{\bar{A}} p_{\bar{A}})$
$\boldsymbol{z} = \bar{\boldsymbol{A}} \boldsymbol{b}_{\boldsymbol{i}}$
vector unpacking, $O(n)$
for $\tilde{i} = 1 : n$
  $\hat{\boldsymbol{y}}_{\boldsymbol{i}}(\tilde{i}) = \boldsymbol{z}(\boldsymbol{k}_{\bar{A}}(\tilde{i}))$

(a) $\boldsymbol{X_i b_i} = \boldsymbol{A b_i}$, where $\boldsymbol{X_i}$ is a singleton with packed storage $(\bar{\boldsymbol{A}},\ \boldsymbol{k}_{\bar{A}},\ m_{\bar{A}},\ p_{\bar{A}})$. FLOP count for each computation step is given in gray text, and the algorithm involves $O(n) + O(m_{\bar{A}} p_{\bar{A}})$ FLOP.

initialize a length-n vector of zeros, O(n)
$\hat{\boldsymbol{y}}_{\boldsymbol{i}} = \text{zeros}(n)$
for $k = 1 : p_{\dot{\boldsymbol{A}}}$
  discrete matrix-vector multiplication, $O(n) + O(m_{\bar{A}_s} p_{\bar{A}_s})$
  compute $\boldsymbol{z} = \boldsymbol{A_s b_i}((k p_{\bar{A}_s} - p_{\bar{A}_s} + 1) : k p_{\bar{A}_s})$ using algorithm in Figure 8.6a
  update $\boldsymbol{z}$ and $\hat{\boldsymbol{y}}_{\boldsymbol{i}}$, $O(sn)$
  update $\boldsymbol{z} = \texttt{diag}(\dot{\boldsymbol{A}}(,\ k))\boldsymbol{z}$ using algorithm in Figure 8.3c
  update $\hat{\boldsymbol{y}}_{\boldsymbol{i}} = \hat{\boldsymbol{y}}_{\boldsymbol{i}} + \boldsymbol{z}$

(b) $\boldsymbol{X_i b_i} = (\dot{\boldsymbol{A}} \mathbin{\widetilde{\otimes}} \boldsymbol{A_s}) \boldsymbol{b_i}$, where $\boldsymbol{X_i}$ is a tensor with s packed margins from $(\bar{\boldsymbol{A}}_{\boldsymbol{1}},\ \boldsymbol{k}_{\bar{A}_1},\ m_{\bar{A}_1},\ p_{\bar{A}_1})$ to $(\bar{\boldsymbol{A}}_{\boldsymbol{s}},\ \boldsymbol{k}_{\bar{A}_s},\ m_{\bar{A}_s},\ p_{\bar{A}_s})$, and $\dot{\boldsymbol{A}} = \boldsymbol{A_1} \mathbin{\widetilde{\otimes}} \boldsymbol{A_2} \mathbin{\widetilde{\otimes}} \cdots \mathbin{\widetilde{\otimes}} \boldsymbol{A_{s-1}}$. FLOP count for each computation step is given in gray text, and the algorithm involves $O(snp_{\dot{\boldsymbol{A}}}) + O(m_{\bar{A}_s} p_{\dot{\boldsymbol{A}}} p_{\bar{A}_s})$ FLOP.

Figure 8.6: Computation of $\boldsymbol{X_i b_i}$ when $\boldsymbol{X_i}$ is a singleton or tensor.

thus an iterative algorithm as in Figure 8.6b applies, and it is $O(\min\{p_{\bar{A}_s}/s,\ n/m_{\bar{A}_s}\})$ times faster than conventional computation.

With centring constraint, we need to compute $\boldsymbol{X_i Q_i^\perp b_i}$. By first obtaining $\boldsymbol{b_i^*} = \boldsymbol{Q_i^\perp b_i}$, $\boldsymbol{X_i b_i^*}$ can be computed with algorithms in Figure 8.6a or 8.6b. $\boldsymbol{b_i^*}$ can be conveniently computed by $\boldsymbol{b_i^*} = (\boldsymbol{Q_i}\ \boldsymbol{Q_i^\perp}) \binom{0}{\boldsymbol{b_i}} = \boldsymbol{H_1} \binom{0}{\boldsymbol{b_i}}$, where padding a zero to $\boldsymbol{b_i}$ facilitates the use of Householder transformation.

8.6 Caching for computation of $\boldsymbol{X'WX}$

In §5.2.3 I have explained how a block algorithm or data caching is important for high-performance computing. It is possible to utilize data caching in the matrix weighting step of algorithms in Figures 8.4b, 8.5a and 8.5b to achieve higher performance when it comes to realistic computing on a modern computer with CPU cache. Common weight matrices include

- a diagonal matrix $\boldsymbol{W} = \texttt{diag}(\boldsymbol{w})$ where only the main diagonal $\boldsymbol{w}$ needs be stored;
- a symmetric tri-diagonal matrix, for example the precision matrix of an AR(1) process, like (2.1) or (6.1). Storage for such matrix only requires a vector $\boldsymbol{w}$ of $(2n-1)$ elements by storing the main diagonal and a subdiagonal.

Let $\boldsymbol{Y}$ be an $n \times p_{\boldsymbol{Y}}$ dense matrix, three types of matrix weighting will be met in these algorithms, namely

- matrix row-scaling that overwrites an existing matrix $\boldsymbol{Y} = \texttt{diag}(\boldsymbol{w})\boldsymbol{Y}$;
- matrix row-scaling that writes to a new matrix $\tilde{\boldsymbol{Y}} = \texttt{diag}(\boldsymbol{w})\boldsymbol{Y}$;

Table 8.1: Practical performance of three types of matrix weighting with and without caching. Experiment is taken on an Intel Core i5-2557M Sandy Bridge laptop (see Appendix A for hardware information) with a 400000×300 matrix $\boldsymbol{Y}$, and execution time (in seconds) for 100 replications is given.

	$\boldsymbol{Y} = \texttt{diag}(\boldsymbol{w})\boldsymbol{Y}$	$\tilde{\boldsymbol{Y}} = \texttt{diag}(\boldsymbol{w})\boldsymbol{Y}$	$\boldsymbol{Y} = \boldsymbol{W}\boldsymbol{Y}$
no caching	33.6	41.4	61.7
caching	22.0	33.7	38.4

- symmetric tri-diagonal matrix weighting that overwrites an existing matrix $\tilde{\boldsymbol{Y}} = \boldsymbol{W}\boldsymbol{Y}$, where $\boldsymbol{W}$ is a symmetric tri-diagonal matrix.

Table 8.1 is a practical example on how caching speeds up computation. In the following I will explain how caching is attained for these three types of matrix weighting.

Consider first $\boldsymbol{Y} = \texttt{diag}(\boldsymbol{w})\boldsymbol{Y}$. Conventional computation does column-wise scaling (see algorithm in Figure 8.7a). If n is small enough such that $\boldsymbol{w}$ and $\boldsymbol{Y}(,\ j)$ fit in L1 CPU cache, $\boldsymbol{w}$ only needs be fetched from RAM once, and it stays in cache until all columns of $\boldsymbol{Y}$ are processed. The total number of reads from RAM is $(np_{\boldsymbol{Y}} + n)$, including $np_{\boldsymbol{Y}}$ reads for $\boldsymbol{Y}$ and n reads for $\boldsymbol{w}$. However, if n is big, $\boldsymbol{w}$ has to be streamed from RAM for every column of $\boldsymbol{Y}$, as $\boldsymbol{w}$ would be evicted from the cache after a column of $\boldsymbol{Y}$ is processed. This results in $2np_{\boldsymbol{Y}}$ total number of reads from RAM. The amount of RAM access has a noticeable impact on performance, because the operation only involves $np_{\boldsymbol{Y}}$ floating-point multiplication, so it is memory-bound rather than computation-bound. To cache $\boldsymbol{w}$ for arbitrary n, matrix row-scaling can be done tile-by-tile, as demonstrated by the algorithm in Figure 8.7b. In theory, tile size c should be chosen so that $\boldsymbol{w}(h)$ occupies about half of the cache (because another half is reserved for a column of $\boldsymbol{Y}$). Most computers today have 32KB L1 cache, which can hold 4096 double precision floating-point numbers. This means that $c = 2000$ is near optimal. However, in practice it seems that performance is not very sensitive to the exact choice of c; any values between 200 and 4000 give about the same performance. But if for example, $c = 100$, the execution time becomes 26.4 seconds.

For $\tilde{\boldsymbol{Y}} = \texttt{diag}(\boldsymbol{w})\boldsymbol{Y}$, the same kind of tiling can be done, except that the theoretically optimal tile size is chosen so that $\boldsymbol{w}(h)$ occupies 1/3 of cache (because a column of $\tilde{\boldsymbol{Y}}$ and $\boldsymbol{Y}$ both require 1/3 of cache, too), giving $c = 1350$. For this operation, performance is more sensitive to c. If for example, $c = 800$ or $c = 1600$, exeuction time becomes about 36.6 seconds.

Now consider $\boldsymbol{Y} = \boldsymbol{W}\boldsymbol{Y}$, or

$$\begin{pmatrix} \boldsymbol{w}(1) & \boldsymbol{w}(2) & & & & \\ \boldsymbol{w}(2) & \boldsymbol{w}(3) & \boldsymbol{w}(4) & & & \\ & \ddots & \ddots & \ddots & & \\ & & \boldsymbol{w}(2n-4) & \boldsymbol{w}(2n-3) & \boldsymbol{w}(2n-2) & \\ & & & \boldsymbol{w}(2n-2) & \boldsymbol{w}(2n-1) \end{pmatrix} \begin{pmatrix} \cdots & \boldsymbol{Y}(\ 1,\ j) & \cdots \\ \cdots & \boldsymbol{Y}(\ 2,\ j) & \cdots \\ & \vdots & \\ \cdots & \boldsymbol{Y}(n-1,\ j) & \cdots \\ \cdots & \boldsymbol{Y}(\ n,\ j) & \cdots \end{pmatrix},$$

with a packed storage for $\boldsymbol{W}$. Conventional computation does column-wise matrix-vector multiplication (see algorithm in Figure 8.7c), but if n is too big for $\boldsymbol{w}$ and $\boldsymbol{Y}(,\ j)$ to fit in cache, $\boldsymbol{w}$ has to be streamed from RAM for every column of $\boldsymbol{Y}$. We can cache $\boldsymbol{w}$ by row tiling, but a working vector $\boldsymbol{u}$ is needed as a buffer to store a row of $\boldsymbol{Y}$. Algorithm in Figure 8.7d implements this. Caching has a even more pronounced effect on this operation than matrix row-scaling, because it reduces total amount of RAM access from $3np_{\boldsymbol{Y}}$ to $(np_{\boldsymbol{Y}} + 2n - 1)$. As $(2c - 1)$ elements of $\boldsymbol{w}$ and c elements of $\boldsymbol{Y}$ would be accessed in the innermost $\tilde{i}$-loop, c should be chosen so that $3c$ double precision floating-point numbers can fit in cache. With a 32KB L1 cache, this is $c = 1350$. But once again, performance is insensitive to c in practice, as long as it is not too small (say 100) or too big (say 4000).

```
for j = 1 : p_Y
    Y(:, j) = w ∘ Y(:, j)
```

(a) uncached matrix row-scaling $\boldsymbol{Y} = \texttt{diag}(\boldsymbol{w})\boldsymbol{Y}$

```
i = 1
while (i ≤ n)
    c̃ = min{c, n − i + 1}
    row-scaling of Y(i : (i + c̃ − 1), :)
    Y(i : (i + c̃ − 1), :) = diag(w(i : (i + c̃ − 1)))Y(i : (i + c̃ − 1), :)
    i = i + c̃
```

(b) cached matrix row-scaling $\boldsymbol{Y} = \texttt{diag}(\boldsymbol{w})\boldsymbol{Y}$

```
for j = 1 : p_Y
    update Y(1, j)
    r = Y(1, j)
    Y(1, j) = w(1) · r + w(2) · Y(2, j)
    update Y(2 : (n − 1), j)
    for i = 2 : (n − 1)
        s = w(2i − 2) · r + w(2i − 1) · Y(i, j) + w(2i) · Y(i + 1, j)
        r = Y(i, j)
        Y(i, j) = s
    update Y(n, j)
    Y(n, j) = w(2n − 2) · r + w(2n − 1) · Y(n, j)
```

(c) uncached tri-diagonal weighting $\boldsymbol{Y} = \boldsymbol{W}\boldsymbol{Y}$

```
initialize a length-p_Y buffer vector u
u = zeros(p_Y)
update Y(1, :)
for j = 1 : p_Y
    u(j) = Y(1, j)
    Y(1, j) = w(1) · u(j) + w(2) · Y(2, j)
update Y(2 : (n − 1), j), one tile per time
i = 2
while (i ≤ (n − 1))
    c̃ = min{c, n − i}
    for j = 1 : p_Y
        r = u(j)
        for ĩ = i : (i + c̃ − 1)
            s = w(2ĩ − 2) · r + w(2ĩ − 1) · Y(ĩ, j) + w(2ĩ) · Y(ĩ + 1, j)
            r = Y(ĩ, j)
            Y(ĩ, j) = s
        u(j) = r
    i = i + c̃
update Y(n, :)
for j = 1 : p_Y
    Y(n, j) = w(2n − 2) · u(j) + w(2n − 1) · Y(n, j)
```

(d) cached tri-diagonal weighting $\boldsymbol{Y} = \boldsymbol{W}\boldsymbol{Y}$

Figure 8.7: Algorithms for matrix weighting with and without caching.

8.7 Experiment on daily logBS model

To demonstrate that algorithms elaborated in previous sections yield significant speedup to pseudo QR reduction and $\hat{\boldsymbol{y}}$ computation in practice, consider their application to the test model (model 7.1) with 1342159 data and 22571 coefficients. A quick test with an arbitrary length-1342159 vector $\boldsymbol{y}$ and an arbitrary length-22571 vector $\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}$ shows that computations of $\boldsymbol{X'Wy}$ and $\hat{\boldsymbol{y}} = \boldsymbol{X}\hat{\boldsymbol{\beta}}_{\boldsymbol{\lambda}}$ respectively take 1.5 and 2.5 seconds in serial computing. By contrast, computing $\hat{\boldsymbol{y}}$ with "online" algorithm and the fast model matrix formation method described in §8.1 still take 3 minutes using 16 threads.

Computation of $\boldsymbol{X'WX}$ takes 11 hours with single-threaded OpenBLAS if using the "online" pseudo QR reduction algorithm. With 16-threaded OpenBLAS it takes 50 minutes. Discrete computation of $\boldsymbol{X'WX}$ takes 5 hours 40 minutes in serial computing, and 34 minutes in 16-threaded parallel computing. Here, multi-threaded parallel computing for $\boldsymbol{X'WX}$ is achieved by parallelizing all steps of the algorithms via OpenMP.

- The matrix row-aggregation in Figure 8.4a can process different columns in parallel;
- With the caching algorithm in Figure 8.7b, the matrix row-scaling in Figure 8.4b can process different row tiles in parallel;
- The column unpacking of $\boldsymbol{B}$ in Figure 8.5a and 8.5b can process different columns in parallel;
- The matrix weighting in Figure 8.5a and 8.5b is also parallelizable. For a diagonal $\boldsymbol{W}$, this is just a parallel matrix row-scaling. For a symmetric tri-diagonal $\boldsymbol{W}$, the caching algorithm in Figure 8.7d implies that there is dependency between different row tiles through the buffer vector. However, we can instead split $\boldsymbol{Y}$ into a few column chunks, and weight different chunks in parallel.

All these computations are memory-bound, so parallel scaling is up to how well different CPUs in a machine can do parallel read from RAM or parallel write to RAM. Different machines may have different front-side bus (FSB) bandwidth, thus the scaling factor for these operations are likely to vary from machine to machine. On an Intel Xeon E5-2650 v2 workstation, parallel scaling is not too bad, with 5.5 times speedup at 8 threads and 10 times speedup at 16 threads.

8.8 Summary

This Chapter has developed algorithms for computing matrix cross-product $\boldsymbol{X'WX}$ when the sub-matrices of $\boldsymbol{X}$ corresponding to individual GAM terms in a linear predictor $\boldsymbol{\eta} = \boldsymbol{X_0\beta_0} + \sum_{i=1}^{l} \boldsymbol{X_i\beta_i}$ are stored in a packed storage format from covariate discretization. Computation of each $\boldsymbol{X_i'WX_j}$ is $O(p)$ times faster than the "online" pseudo QR reduction algorithm, where p is the number of columns in $\boldsymbol{X_i}$ if $\boldsymbol{X_i}$ is a singleton, and that of the last margin of $\boldsymbol{X_i}$ if $\boldsymbol{X_i}$ is a tensor.

This idea of covariate discretization was first introduced in Lang et al. (2014) to obtain efficient storage and computation for large datasets. However, in that paper they employ smooths of one covariate and only require terms of the form $\boldsymbol{X_i'WX_i}$, but not $\boldsymbol{X_i'WX_j}$. So methods in this Chapter have extended the idea by

- discretizing covariates in multivariate splines (say tensor product splines) and exploiting the resulting packed storage for computation;
- developing algorithms for computing any $\boldsymbol{X_i'WX_j}$.

Regarding the first point, method in this Chapter discretizes covariates marginally rather than jointly, so that spline functions can be evaluated at their unique combinations. The discretization result is also stored marginally rather than jointly, which substantially reduces the storage complexity. This is a major advantage over Helwig and Ma (2016). For example, consider a two margin tensor product spline. To store s unique covariate values of the first margin and t unique covariate values of the second margin, the marginal approach only requires $(s + t)$ storage costs, but the joint approach requires st storage costs. As a result, the joint approach has more difficulty in balancing storage costs and precision of discretization.

A hard question is how to interpret "Computation of each $\boldsymbol{X_i'WX_j}$ is $O(p)$ times faster than the "online" pseudo QR reduction algorithm". Indeed, this does not seem to agree with the benchmarking result in §8.7, where the new discrete computation method is only two times as fast as the old "online" algorithm. The problem is, that $O(p)$ factor is derived from comparing FLOP count of the two computation methods, but in reality FLOP count is not everything. The "online" pseudo QR reduction, while computationally more expensive, can attain high performance computing with the aid of an optimized BLAS. The new discrete method, while computationally cheaper, is not rich in level-3 BLAS operations. On the contrary, it is rich in vector-oriented operations like unpacking, aggregation, reweighting, etc. These operations are like level-2 BLAS operations; they are memory-bound rather than CPU-bound. In §8.6, some attempts were made to improve the performance of this new computation method, however, just like a matrix-vector multiplication can hardly be boosted by a block algorithm or data caching (see a relevant discussion on this in §5.2.3), the practical gains is very limited.

Is there any way, to overcome this drawback of the discrete computation method? Yes, there is. It is `bamboos`, a new computational engine for discrete $\boldsymbol{X'WX}$. Welcome to the next Chapter, the most exciting Chapter of this thesis.

Chapter 9

Bamboos: boosting discrete pseudo QR reduction by another magnitude

In this Chapter, I will introduce `bamboos` (`bam` booster), a new computational engine for matrix cross-product $\boldsymbol{X'WX}$ that is truely high performance computing, able to boost the discrete computation method developed in the previous Chapter by another magnitude. This Chapter is laid out as follows.

- §9.1 explains the basic idea of `bamboos`;
- §9.2 makes FLOP count comparison between `bamboos` and the old discrete algorithms;
- §9.3 explains how `bamboos` actually achieves high performance;
- §9.4 benchmarks `bamboos` and the old discrete algorithms by fitting the test daily logBS model, model 7.1;
- §9.5 summarizes key features and advantages of `bamboos`.

9.1 A basic description of `bamboos` algorithms

Recall that for a linear predictor $\boldsymbol{\eta} = \boldsymbol{X_0\beta_0} + \sum_{i=1}^{l} \boldsymbol{f_i} = \boldsymbol{X_0\beta_0} + \sum_{i=1}^{l} \boldsymbol{X_i\beta_i}$ with a design matrix $\boldsymbol{X_0}$ for parametric model terms and design matrices $\boldsymbol{X_i}$'s for splines $\boldsymbol{f_i}$'s, algorithms in §8.4 can be interpreted as such:

- For any $\boldsymbol{X_i'WX_0}$, treat $\boldsymbol{X_0}$ as $\boldsymbol{Y}$, then compute $\boldsymbol{X_i'WY}$ using algorithms in Figures 8.4a and 8.4b;
- for any $\boldsymbol{X_i'WX_j}$ $(i \geq j > 0)$, unpack $\boldsymbol{X_j}$ from its packed storage into a full design matrix $\boldsymbol{Y}$, then compute $\boldsymbol{X_i'WY}$ using algorithms in Figures 8.4a and 8.4b.

So computation of $\boldsymbol{X_i'WY}$ is elementary. By contrast, in `bamboos`, such computation is only useful for computing $\boldsymbol{X_i'WX_0}$. For other blocks, computation of $\boldsymbol{X_i'WX_j}$ between two singletons plays a pivotal role, as $\boldsymbol{X_i'WX_j}$ involving tensors would be cast into its iterative application. As it will be shown, there exists algorithms for computing a "singleton $\times$ singleton" block without matrix unpacking, hence `bamboos` algorithms require no matrix unpacking at all! In the rest of this section,

- §9.1.1 motivates algorithms for "singleton $\times$ singleton" blocks;

- §9.1.2 extends these algorithms for "singleton × Tensor", "Tensor × singleton" and "Tensor × Tensor" blocks;
- §9.1.3 is a consolidation, summarizing above algorithms with a unified representation.

In the forthcoming demonstration, I will adopt the same notations used in Chapter 8. For singletons $\boldsymbol{X_i}$ and $\boldsymbol{X_j}$, denote that they respectively have packed storage $(\bar{\boldsymbol{A}},\ \boldsymbol{k}_{\bar{A}},\ m_{\bar{A}},\ p_{\bar{A}})$ and $(\bar{\boldsymbol{B}},\ \boldsymbol{k}_{\bar{B}},\ m_{\bar{B}},\ p_{\bar{B}})$. For tensors $\boldsymbol{X_i}$ and $\boldsymbol{X_j}$, denote that they respectively have s packed margins from $(\bar{\boldsymbol{A}}_{\boldsymbol{1}},\ \boldsymbol{k}_{\bar{A}_1},\ m_{\bar{A}_1},\ p_{\bar{A}_1})$ to $(\bar{\boldsymbol{A}}_{\boldsymbol{s}},\ \boldsymbol{k}_{\bar{A}_s},\ m_{\bar{A}_s},\ p_{\bar{A}_s})$, and t packed margins from $(\bar{\boldsymbol{B}}_{\boldsymbol{1}},\ \boldsymbol{k}_{\bar{B}_1},\ m_{\bar{B}_1},\ p_{\bar{B}_1})$ to $(\bar{\boldsymbol{B}}_{\boldsymbol{t}},\ \boldsymbol{k}_{\bar{B}_t},\ m_{\bar{B}_t},\ p_{\bar{B}_t})$. I also define $\dot{\boldsymbol{A}} = \boldsymbol{A_1} \bar{\otimes} \boldsymbol{A_2} \bar{\otimes} \cdots \boldsymbol{A_{s-1}}$ with $p_{\dot{A}} = \prod_{i=1}^{s-1} p_{\bar{A}_i}$, and $\dot{\boldsymbol{B}} = \boldsymbol{B_1} \bar{\otimes} \boldsymbol{B_2} \bar{\otimes} \cdots \boldsymbol{B_{t-1}}$ with $p_{\dot{B}} = \prod_{i=1}^{t-1} p_{\bar{B}_i}$, so that any tensor can be expressed with two margins.

9.1.1 $\boldsymbol{X_i'WX_j}$ between singletons

Consider first a diagonal weight matrix $\boldsymbol{W} = \texttt{diag}(\boldsymbol{w})$, then $\boldsymbol{X_i'WX_i} = \boldsymbol{A'WA}$ and $\boldsymbol{X_i'WX_j} = \boldsymbol{A'WB}$ can be computed using algorithms in Figure 9.1a. Note that unlike the discrete algorithms in the last Chapter where weights are absorbed into the design matrix on the right and processed together with a matrix, here, weights are processed independently and are aggregated into a contingency table / matrix $\bar{\boldsymbol{W}}$. Below is a mathematical proof of the algorithm for $\boldsymbol{X_i'WX_j} = \boldsymbol{A'WB}$. Partition $\boldsymbol{A}$ and $\boldsymbol{B}$ by rows, it can then be recognized that $\boldsymbol{A'WB}$ is a quadratic form:

$$\boldsymbol{A'WB} = \begin{bmatrix}\boldsymbol{A}'(1,\ ) & \boldsymbol{A}'(2,\ ) & \cdots & \boldsymbol{A}'(n,\ )\end{bmatrix} \begin{bmatrix} w_1 & & & \\ & w_2 & & \\ & & \ddots & \\ & & & w_n \end{bmatrix} \begin{bmatrix}\boldsymbol{B}(1,\ ) \\ \boldsymbol{B}(2,\ ) \\ \vdots \\ \boldsymbol{B}(n,\ )\end{bmatrix} = \sum_{i=1}^{n} w_i \boldsymbol{A}'(i,\ )\boldsymbol{B}(i,\ ).$$

Now I am going to tweak the summation to sum over unique rows of $\boldsymbol{A}$ and $\boldsymbol{B}$, i.e., rows of packed design matrices $\bar{\boldsymbol{A}}$ and $\bar{\boldsymbol{B}}$. Essentially it is grouping w_i by $(\boldsymbol{k}_{\bar{A}}(i),\ \boldsymbol{k}_{\bar{B}}(i))$ pairs, giving:

$$\sum_{u=1}^{p_{\bar{A}}} \sum_{v=1}^{p_{\bar{B}}} \left(\sum_{\substack{\boldsymbol{k}_{\bar{A}}(i)=u \\ \boldsymbol{k}_{\bar{B}}(i)=v}} w_i \right) \bar{\boldsymbol{A}}'(u,\ )\bar{\boldsymbol{B}}(v,\ ) := \sum_{u=1}^{p_{\bar{A}}} \sum_{v=1}^{p_{\bar{B}}} \bar{w}_{uv} \bar{\boldsymbol{A}}'(u,\ )\bar{\boldsymbol{B}}(v,\ ) = \bar{\boldsymbol{A}}'\bar{\boldsymbol{W}}\bar{\boldsymbol{B}},$$

where $\bar{\boldsymbol{W}}(u,\ v) = w_{uv}$ is the group sum, which is what the in the algorithm computes.

The advantage of these algorithms is most obvious for $\boldsymbol{X_i'WX_i}$, whose leading order FLOP count is $O(n) + O(m_{\bar{A}} p_{\bar{A}}^2)$. By contrast, direct computation that first unpacks $\bar{\boldsymbol{A}}$ for $\boldsymbol{A}$, then performs row-scaling $\tilde{\boldsymbol{A}} = \boldsymbol{WA}$ and finally computes matrix cross-product $\boldsymbol{A}'\tilde{\boldsymbol{A}}$ involves $O(np_{\bar{A}}) + O(np_{\bar{A}}) + O(np_{\bar{A}}^2)$ FLOP. So the algorithm is very efficient when $n \gg m_{\bar{A}}$.

For $\boldsymbol{X_i'WX_j}$, the matrix product and cross-product can be computed in two different orders, and the least expensive one should be chosen in practice. But the advantage of the algorithm is not immediately clear, as while $n \gg m_{\bar{A}}$ and $n \gg m_{\bar{B}}$ may hold, it is not guaranteed that $n \gg m_{\bar{A}} m_{\bar{B}}$. For example, consider $n = 10^5$ with $m_{\bar{A}} = m_{\bar{B}} = 10^3$. So the $O(m_{\bar{A}} m_{\bar{B}} p_{\bar{B}})$ or $O(m_{\bar{A}} m_{\bar{B}} p_{\bar{A}})$ complexity for the matrix product can be very expensive, as $m_{\bar{A}} m_{\bar{B}}$ approaches or even goes beyond n. However, since $\bar{\boldsymbol{W}}$ is obtained from n data, it at most has n non-zero entries. As $m_{\bar{A}} m_{\bar{B}}$ becomes increasingly large, $\bar{\boldsymbol{W}}$ becomes increasingly sparse, so I can avoid forming $\bar{\boldsymbol{W}}$ as a dense matrix and do sparse matrix multiplication instead, which only involves $O(Kp_{\bar{B}})$ or $O(Kp_{\bar{A}})$ FLOP, where K is the number of non-zero entries in $\bar{\boldsymbol{W}}$ (sparse $\bar{\boldsymbol{W}}$ will be covered in details in §9.3). Obviously, $K < \min\{m_{\bar{A}} m_{\bar{B}},\ n\}$, so its computational complexity is bounded. In the worst case, the algorithm has a leading order FLOP count of $O(n) + O(np_{\bar{B}}) + O(m_{\bar{A}} p_{\bar{A}} p_{\bar{B}})$ or $O(n) + O(np_{\bar{A}}) + O(m_{\bar{B}} p_{\bar{A}} p_{\bar{B}})$, yet this is still a lot more efficient than direct computation that first unpacks $\bar{\boldsymbol{A}}$ and $\bar{\boldsymbol{B}}$ to $\boldsymbol{A}$ and $\boldsymbol{B}$ (with $O(np_{\bar{A}}) + O(np_{\bar{B}})$ FLOP), then performs row-scaling $\tilde{\boldsymbol{A}} = \boldsymbol{WA}$ or $\tilde{\boldsymbol{B}} = \boldsymbol{WB}$ (with $O(np_{\bar{A}})$ or $O(np_{\bar{B}})$ FLOP)

initialize a length-$m_{\bar{A}}$ vector of zeros	initialize a $m_{\bar{A}} \times m_B$ matrix of zeros
$\bar{\boldsymbol{w}} = \text{zeros}(m_{\bar{A}})$	$\bar{\boldsymbol{W}} = \text{zeros}(m_{\bar{A}},\ m_{\bar{B}})$
one-way contingency table, $O(n)$	two-way contingency table, $O(n)$
for $i = 1 : n$	for $i = 1 : n$
$\quad \bar{\boldsymbol{w}}(\boldsymbol{k}_{\bar{A}}(i)) \mathrel{+}= \boldsymbol{w}(i)$	$\quad \bar{\boldsymbol{W}}(\boldsymbol{k}_{\bar{A}}(i),\ \boldsymbol{k}_{\bar{B}}(i)) \mathrel{+}= \boldsymbol{w}(i)$
matrix row-scaling, $O(m_{\bar{A}} p_{\bar{A}})$	matrix product, $O(m_{\bar{A}} m_{\bar{B}} p_{\bar{B}})$ or $O(m_{\bar{A}} m_{\bar{B}} p_{\bar{A}})$
$\tilde{\boldsymbol{A}} = \texttt{diag}(\bar{\boldsymbol{w}})\bar{\boldsymbol{A}}$	$\tilde{\boldsymbol{B}} = \bar{\boldsymbol{W}}\bar{\boldsymbol{B}}$ or $\tilde{\boldsymbol{A}} = \boldsymbol{A}'\bar{\boldsymbol{W}}$
matrix cross-product, $O(m_{\bar{A}} p_{\bar{A}}^2)$	cross-product, $O(m_{\bar{A}} p_{\bar{A}} p_{\bar{B}})$ or $O(m_{\bar{B}} p_{\bar{A}} p_{\bar{B}})$
$\boldsymbol{X}_i'\boldsymbol{W}\boldsymbol{X}_i = \bar{\boldsymbol{A}}'\tilde{\boldsymbol{A}}$	$\boldsymbol{X}_i'\boldsymbol{W}\boldsymbol{X}_j = \bar{\boldsymbol{A}}'\tilde{\boldsymbol{B}}$ or $\tilde{\boldsymbol{A}}\bar{\boldsymbol{B}}$

(a) Computation of $\boldsymbol{X}_i'\boldsymbol{W}\boldsymbol{X}_i = \boldsymbol{A}'\boldsymbol{W}\boldsymbol{A}$ and $\boldsymbol{X}_i'\boldsymbol{W}\boldsymbol{X}_j = \boldsymbol{A}'\boldsymbol{W}\boldsymbol{B}$, where $\boldsymbol{W} = \texttt{diag}(\boldsymbol{w})$ is a diagonal matrix. Note that if matrix row-scaling is performed by $\tilde{\boldsymbol{A}} = \texttt{diag}(\sqrt{\boldsymbol{w}})\bar{\boldsymbol{A}}$, the matrix cross-product $\boldsymbol{X}_i'\boldsymbol{W}\boldsymbol{X}_i = \tilde{\boldsymbol{A}}'\tilde{\boldsymbol{A}}$ will involve 50% less FLOP count. However, This idea does not work when there are negative weights in $\boldsymbol{w}$. This can happen in two cases: 1) the working weights in P-IRLS (see algorithm in Figure 4.12) can contain negative values; 2) the weights can be negative after being reweighted (see algorithms in the next section).

initialize a $m_{\bar{A}} \times m_{\bar{B}}$ matrix of zeros
$\bar{\boldsymbol{W}} = \text{zeros}(m_{\bar{A}},\ m_{\bar{B}})$
two-way contingency table, $O(n)$
for $i = 1 : n$
$\quad \bar{\boldsymbol{W}}(\boldsymbol{k}_{\bar{A}}(i),\ \boldsymbol{k}_{\bar{B}}(i)) \mathrel{+}= \boldsymbol{w}_0(i)$
for $i = 1 : (n-1)$
$\quad \bar{\boldsymbol{W}}(\boldsymbol{k}_{\bar{A}}(i),\ \boldsymbol{k}_{\bar{B}}(i+1)) \mathrel{+}= \boldsymbol{w}_1(i)$
for $i = 1 : (n-1)$
$\quad \bar{\boldsymbol{W}}(\boldsymbol{k}_{\bar{A}}(i+1),\ \boldsymbol{k}_{\bar{B}}(i)) \mathrel{+}= \boldsymbol{w}_{-1}(i)$
matrix product, $O(m_{\bar{A}} m_{\bar{B}} p_{\bar{B}})$ or $O(m_{\bar{A}} m_{\bar{B}} p_{\bar{A}})$
$\tilde{\boldsymbol{B}} = \bar{\boldsymbol{W}}\bar{\boldsymbol{B}}$ or $\tilde{\boldsymbol{A}} = \bar{\boldsymbol{A}}'\bar{\boldsymbol{W}}$
matrix cross-product, $O(m_{\bar{A}} p_{\bar{A}} p_{\bar{B}})$ or $O(m_{\bar{B}} p_{\bar{A}} p_{\bar{B}})$
$\boldsymbol{X}_i'\boldsymbol{W}\boldsymbol{X}_j = \bar{\boldsymbol{A}}'\tilde{\boldsymbol{B}}$ or $\tilde{\boldsymbol{A}}\bar{\boldsymbol{B}}$

(b) Computation of $\boldsymbol{X}_i'\boldsymbol{W}\boldsymbol{X}_j = \boldsymbol{A}'\boldsymbol{W}\boldsymbol{B}$, where $\boldsymbol{W}$ is a tri-diagonal matrix (either symmetric or asymmetric) whose main diagonal, upper subdiagonal and lower subdiagonal are stored as $\boldsymbol{w}_0$, $\boldsymbol{w}_1$ and $\boldsymbol{w}_{-1}$ respectively. See main text on why computation of $\boldsymbol{X}_i'\boldsymbol{W}\boldsymbol{X}_i$ is not provided.

Figure 9.1: Computation of $\boldsymbol{X}_i'\boldsymbol{W}\boldsymbol{X}_j$ when both $\boldsymbol{X}_i$ and $\boldsymbol{X}_j$ are singletons, respectively with packed storage $(\bar{\boldsymbol{A}},\ \boldsymbol{k}_{\bar{A}},\ m_{\bar{A}},\ p_{\bar{A}})$ and $(\bar{\boldsymbol{B}},\ \boldsymbol{k}_{\bar{B}},\ m_{\bar{B}},\ p_{\bar{B}})$. FLOP count for each computation step is given in gray text. Note that the result for matrix product is given assuming that $\bar{\boldsymbol{W}}$ is (constructed as) dense, therefore the FLOP count is proportional to its size $m_{\bar{A}} m_{\bar{B}}$. But $\bar{\boldsymbol{W}}$ can be sparse with $K < \min\{m_{\bar{A}} m_{\bar{B}},\ n\}$ (or even $K \ll \min\{m_{\bar{A}} m_{\bar{B}},\ n\}$) non-zero elements. Constructing $\bar{\boldsymbol{W}}$ as sparse and exploiting sparsity in matrix-matrix multiplication in this situation can substantially reduce the $O(m_{\bar{A}} m_{\bar{B}} p_{\bar{B}})$ or $O(m_{\bar{A}} m_{\bar{B}} p_{\bar{A}})$ complexity to $O(K p_{\bar{B}})$ or $O(K p_{\bar{A}})$. Details on sparse construction of $\bar{\boldsymbol{W}}$ will be given later in §9.3.

and finally computes the matrix cross-product $\tilde{\boldsymbol{A}}'\boldsymbol{B}$ or $\boldsymbol{A}'\tilde{\boldsymbol{B}}$ (with $O(n p_{\bar{A}} p_{\bar{B}})$ FLOP) when $n \gg m_{\bar{A}}$ and $n \gg m_{\bar{B}}$.

When $\boldsymbol{W}$ is a tri-diagonal matrix like (2.1) or (6.1), it can be decomposed into three matrices:

$$\begin{pmatrix} \bullet & \bullet & & & \\ \bullet & \bullet & \bullet & & \\ & \bullet & \bullet & \bullet & \\ & & \bullet & \bullet & \bullet \\ & & & \bullet & \bullet \end{pmatrix} = \begin{pmatrix} \bullet & & & & \\ & \bullet & & & \\ & & \bullet & & \\ & & & \bullet & \\ & & & & \bullet \end{pmatrix} + \left(\begin{array}{ccccc} 0 & \bullet & & & \\ 0 & & \bullet & & \\ 0 & & & \bullet & \\ 0 & & & & \bullet \\ \hline 0 & 0 & 0 & 0 & 0 \end{array}\right) + \left(\begin{array}{ccccc} 0 & 0 & 0 & 0 & 0 \\ \bullet & & & & 0 \\ & \bullet & & & 0 \\ & & \bullet & & 0 \\ & & & \bullet & 0 \end{array}\right).$$

Denote the first diagonal matrix by $\boldsymbol{W}_0$, and the diagonal submatrices in the second and third matrices by $\boldsymbol{W}_1$ and $\boldsymbol{W}_{-1}$, respectively. Then for example, $\boldsymbol{X}_i'\boldsymbol{W}\boldsymbol{X}_j$ decomposes into

$$\boldsymbol{X}_i'\boldsymbol{W}_0\boldsymbol{X}_j + \boldsymbol{X}_i(1:(n-1),\ )'\boldsymbol{W}_1\boldsymbol{X}_j(2:n,\ ) + \boldsymbol{X}_i(2:n,\ )'\boldsymbol{W}_{-1}\boldsymbol{X}_j(1:(n-1),\ ),$$

where each part can be computed using algorithms in Figure 9.1a. See Figure 9.1b for an example implementation (and note that the three s can be merged for conciseness). Computation of $\boldsymbol{X}_i'\boldsymbol{W}\boldsymbol{X}_i =$

for $k = 1 : p_{\dot{A}}$
  compute $\boldsymbol{a} = \dot{\boldsymbol{A}}(,\ k)$ using algorithm in Figure 8.3c
  $\boldsymbol{W_k} = \texttt{diag}(\boldsymbol{a})\boldsymbol{W}$
  compute $\boldsymbol{Z} = \boldsymbol{A_s'W_kB}$ using algorithms in Figure 9.1a or 9.1b
  $\boldsymbol{(X_i'WX_j)}((kp_{\bar{A}_s} - p_{\bar{A}_s} + 1) : kp_{\bar{A}_s},\ ) = \boldsymbol{Z}$

(a) Computation of $\boldsymbol{X_i'WX_j} = \boldsymbol{(\dot{A} \widetilde{\otimes} A_s)'WB}$, where $\boldsymbol{X_j}$ is a singleton with packed storage $(\bar{\boldsymbol{B}},\ \boldsymbol{k}_{\bar{B}},\ m_{\bar{B}},\ p_{\bar{B}})$.

for $k = 1 : p_{\dot{A}}$
  compute $\boldsymbol{a} = \dot{\boldsymbol{A}}(,\ k)$ using algorithm in Figure 8.3c
  $\boldsymbol{W_k} = \texttt{diag}(\boldsymbol{a})\boldsymbol{W}$
  compute $\boldsymbol{Z} = \boldsymbol{(\dot{B} \widetilde{\otimes} B_t)'W_k'A_s}$ using algorithm in Figure 9.2a
  $\boldsymbol{(X_i'WX_j)}((kp_{\bar{A}_s} - p_{\bar{A}_s} + 1) : kp_{\bar{A}_s},\ ) = \boldsymbol{Z'}$

(b) Computation of $\boldsymbol{X_i'WX_j} = \boldsymbol{(\dot{A} \widetilde{\otimes} A_s)'W(\dot{B} \widetilde{\otimes} B_t)}$, where $\boldsymbol{X_j}$ is a tensor with t packed margins from $(\bar{\boldsymbol{B}}_{\boldsymbol{1}},\ \boldsymbol{k}_{\bar{B}_1},\ m_{\bar{B}_1},\ p_{\bar{B}_1})$ to $(\bar{\boldsymbol{B}}_{\boldsymbol{t}},\ \boldsymbol{k}_{\bar{B}_t},\ m_{\bar{B}_t},\ p_{\bar{B}_t})$.

Figure 9.2: Computation of $\boldsymbol{X_i'WX_j}$ when $\boldsymbol{X_i}$ is a tensor with s packed margins from $(\bar{\boldsymbol{A}}_{\boldsymbol{1}},\ \boldsymbol{k}_{\bar{A}_1},\ m_{\bar{A}_1},\ p_{\bar{A}_1})$ to $(\bar{\boldsymbol{A}}_{\boldsymbol{s}},\ \boldsymbol{k}_{\bar{A}_s},\ m_{\bar{A}_s},\ p_{\bar{A}_s})$, and $\boldsymbol{X_j}$ is a singleton or a tensor.

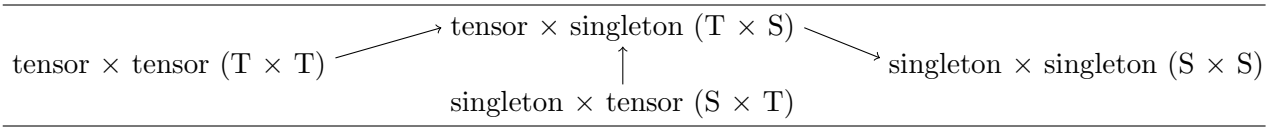


Figure 9.3: Illustrating the computational idea of $\boldsymbol{X_i'WX_j}$ with a dependency graph. Matrix cross-product between two singletons is the elementary computation, and other cases are its iterative application.

$\boldsymbol{A'WA}$ can simply use the same algorithm. The seemingly more efficient computations by first doing a Cholesky factorization $\bar{\boldsymbol{W}} = \boldsymbol{U'U}$, then computing a triangular matrix multiplication $\tilde{\boldsymbol{A}} = \boldsymbol{U}\bar{\boldsymbol{A}}$, and finally taking a matrix cross-product $\boldsymbol{X_iWX_i} = \tilde{\boldsymbol{A}}'\boldsymbol{W}\tilde{\boldsymbol{A}}$ does not generally apply, because $\boldsymbol{W}$ may be asymmetric as will be seen in the next section.

9.1.2 $\boldsymbol{X_i'WX_j}$ with tensors

Assume that $\boldsymbol{X_i}$ is a tenson, then by applying (1.1) to $\dot{\boldsymbol{A}} \widetilde{\otimes} \boldsymbol{A_s}$, there is

$$\boldsymbol{(\dot{A} \widetilde{\otimes} A_s)'WX_j} = \begin{bmatrix} \boldsymbol{A_s'}\texttt{diag}(\dot{\boldsymbol{A}}(,\ 1)) \\ \boldsymbol{A_s'}\texttt{diag}(\dot{\boldsymbol{A}}(,\ 2)) \\ \vdots \\ \boldsymbol{A_s'}\texttt{diag}(\dot{\boldsymbol{A}}(,\ p_{\dot{A}})) \end{bmatrix} \boldsymbol{WX_j} = \begin{bmatrix} \boldsymbol{A_s'}(\texttt{diag}(\dot{\boldsymbol{A}}(,\ 1))\boldsymbol{WX_j}) \\ \boldsymbol{A_s'}(\texttt{diag}(\dot{\boldsymbol{A}}(,\ 2))\boldsymbol{WX_j}) \\ \vdots \\ \boldsymbol{A_s'}(\texttt{diag}(\dot{\boldsymbol{A}}(,\ p_{\dot{A}}))\boldsymbol{WX_j}) \end{bmatrix}.$$

That is, $\boldsymbol{X_i'WX_j}$ can be obtained by computing a matrix cross-product $\boldsymbol{A_s'W_kX_j}$ block by block, where the k^{th} block has a working weight matrix $\boldsymbol{W_k} = \texttt{diag}(\dot{\boldsymbol{A}}(,\ k))\boldsymbol{W}$. Be aware that when $\boldsymbol{W}$ is symmetric tri-diagonal, $\boldsymbol{W_k}$ is asymmetric. Now,

1. if $\boldsymbol{X_j}$ is a singleton, computation of this block can use algorithms in Figure 9.1a or 9.1b, giving an iterative algorithm in Figure 9.2a;
2. if $\boldsymbol{X_j}$ is also a tensor, I can first compute $\boldsymbol{X_j'W_k'A_s}$ using the previous algorithm, then transpose the result. This gives an iterative algorithm in Figure 9.2b.

In summary, such computational idea is illustrated by the dependency graph in Figure 9.3.

for $h = 1 : p_{\dot{B}}$
  compute $\boldsymbol{b} = \boldsymbol{B}(,\ h)$ using algorithm in Figure 8.3c
  $\boldsymbol{W}_{\boldsymbol{.h}} = \boldsymbol{W}\texttt{diag}(\boldsymbol{b})$
  for $g = 1 : p_{\dot{A}}$
    compute $\boldsymbol{a} = \boldsymbol{A}(,\ g)$ using algorithm in Figure 8.3c
    $\boldsymbol{W}_{\boldsymbol{gh}} = \texttt{diag}(\boldsymbol{a})\boldsymbol{W}_{\boldsymbol{.h}}$
    compute $\boldsymbol{Z} = \boldsymbol{A}'_{\boldsymbol{s}}\boldsymbol{W}_{\boldsymbol{gh}}\boldsymbol{B}_{\boldsymbol{t}}$ using algorithms in Figure 9.1a or Figure 9.1b
    $(\boldsymbol{X}'_{\boldsymbol{i}}\boldsymbol{W}\boldsymbol{X}_{\boldsymbol{j}})((gp_{\bar{A}_s} - p_{\bar{A}_s} + 1) : gp_{\bar{A}_s},\ (hp_{\bar{B}_t} - p_{\bar{B}_t} + 1) : hp_{\bar{B}_t}) = \boldsymbol{Z}$

Figure 9.4: A concise presentation of the algorithms in Figure 9.2a and Figure 9.2b. $\boldsymbol{X}_{\boldsymbol{i}}$ is a tensor with s packed margins from $(\bar{\boldsymbol{A}}_{\boldsymbol{1}},\ \boldsymbol{k}_{\bar{A}_1},\ m_{\bar{A}_1},\ p_{\bar{A}_1})$ to $(\bar{\boldsymbol{A}}_{\boldsymbol{s}},\ \boldsymbol{k}_{\bar{A}_s},\ m_{\bar{A}_s},\ p_{\bar{A}_s})$, and $\boldsymbol{X}_{\boldsymbol{j}}$ is a tensor with t packed margins from $(\bar{\boldsymbol{B}}_{\boldsymbol{1}},\ \boldsymbol{k}_{\bar{B}_1},\ m_{\bar{B}_1},\ p_{\bar{B}_1})$ to $(\bar{\boldsymbol{B}}_{\boldsymbol{t}},\ \boldsymbol{k}_{\bar{B}_t},\ m_{\bar{B}_t},\ p_{\bar{B}_t})$. It is legitimate to have $s = 1$ and / or $t = 1$ so that a tensor degenerates to a singleton.

9.1.3 A unified representation of $\boldsymbol{X}'_{\boldsymbol{i}}\boldsymbol{W}\boldsymbol{X}_{\boldsymbol{j}}$ computation

In fact, there is a unified representation for the computation of $\boldsymbol{X}'_{\boldsymbol{i}}\boldsymbol{W}\boldsymbol{X}_{\boldsymbol{j}}$. Note that a singleton can be seen as a one-margin tensor, so I don't have to distinguish a singleton and a tensor when making discussion.

Let $\boldsymbol{X}_{\boldsymbol{i}} = (\dot{\boldsymbol{A}} \mathbin{\widetilde{\otimes}} \boldsymbol{A}_{\boldsymbol{s}})$ and $\boldsymbol{X}_{\boldsymbol{j}} = (\dot{\boldsymbol{B}} \mathbin{\widetilde{\otimes}} \boldsymbol{B}_{\boldsymbol{t}})$, and define $\dot{\boldsymbol{A}} = \boldsymbol{1}$ (a single-column matrix of 1) and $p_{\dot{A}} = 1$ for $s = 1$. Similarly define $\dot{\boldsymbol{B}} = \boldsymbol{1}$ and $p_{\dot{B}} = 1$ for $t = 1$. Then by applying (1.1) to both $\dot{\boldsymbol{A}} \mathbin{\widetilde{\otimes}} \boldsymbol{A}_{\boldsymbol{s}}$ and $\dot{\boldsymbol{B}} \mathbin{\widetilde{\otimes}} \boldsymbol{B}_{\boldsymbol{t}}$, there is

$$(\dot{\boldsymbol{A}} \mathbin{\widetilde{\otimes}} \boldsymbol{A}_{\boldsymbol{s}})'\boldsymbol{W}(\dot{\boldsymbol{B}} \mathbin{\widetilde{\otimes}} \boldsymbol{B}_{\boldsymbol{t}}) = \begin{bmatrix} & \vdots & \\ \cdots & \boldsymbol{A}'_{\boldsymbol{s}}\texttt{diag}(\dot{\boldsymbol{A}}(,\ g))\boldsymbol{W}\texttt{diag}(\dot{\boldsymbol{B}}(,\ h))\boldsymbol{B}_{\boldsymbol{t}} & \cdots \\ & \vdots & \end{bmatrix}.$$

The RHS block matrix has $p_{\dot{A}} \times p_{\dot{B}}$ block elements.

- If $s > 1$ and $t > 1$ (i.e., both $\boldsymbol{X}_{\boldsymbol{i}}$ and $\boldsymbol{X}_{\boldsymbol{j}}$ are tensors), the RHS block matrix has s block rows and t block columns. The weight matrix for the $(g,\ h)^{\text{th}}$ block is $\texttt{diag}(\dot{\boldsymbol{A}}(,\ g))\boldsymbol{W}\texttt{diag}(\dot{\boldsymbol{B}}(,\ h))$.
- If $s > 1$ and $t = 1$ (i.e., $\boldsymbol{X}_{\boldsymbol{i}}$ is a tensor and $\boldsymbol{X}_{\boldsymbol{j}}$ is a singleton), the RHS block matrix has a block column of $p_{\dot{A}}$ block elements. The weight matrix for the g^{th} block reduces to $\texttt{diag}(\dot{\boldsymbol{A}}(,\ g))\boldsymbol{W}$.
- If $s = 1$ and $t > 1$ (i.e., $\boldsymbol{X}_{\boldsymbol{i}}$ is a singleton and $\boldsymbol{X}_{\boldsymbol{j}}$ is a tensor), the RHS block matrix has a block row of $p_{\dot{B}}$ block elements. The weight matrix for the h^{th} block reduces to $\boldsymbol{W}\texttt{diag}(\dot{\boldsymbol{B}}(,\ h))$.
- If $s = t = 1$ (i.e., both $\boldsymbol{X}_{\boldsymbol{i}}$ and $\boldsymbol{X}_{\boldsymbol{j}}$ are singletons), $p_{\dot{A}} = p_{\dot{B}} = 1$ hence the RHS block matrix only has one block element $\boldsymbol{A}'_{\boldsymbol{s}}\boldsymbol{W}\boldsymbol{B}_{\boldsymbol{t}}$;

In this way, algorithms in Figure 9.2a and Figure 9.2b can be concisely presented by algorithm in Figure 9.4 (and there is also no need to do the matrix transpose in Figure 9.2b). The computational idea is now described by Figure 9.5. This idea is easier to programme in practice than that in Figure 9.5, since writing a single routine with a double loop nest is sufficient to handle all S × T, T × S and T × T cases. By contrast, the design in Figure 9.3 requires writing separate routines for these cases. However, in whichever design, $\boldsymbol{X}'_{\boldsymbol{i}}\boldsymbol{W}\boldsymbol{X}_{\boldsymbol{j}}$ of S × S type is the core.

9.2 FLOP comparison with previous discrete algorithms

So far the FLOP count for `bamboos` algorithms has only been given for S × S (see Figure 9.1a and Figure 9.1b). I will now summarize FLOP count for all S × S, S × T, T × S and T × T blocks in

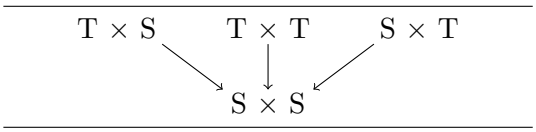


Figure 9.5: Illustrating the computational idea of $\boldsymbol{X_i'WX_j}$ with a dependency graph, following algorithm in Figure 9.4. This design is easier to programme in practice than that in Figure 9.3 because a single routine with a double loop nest is sufficient to handle all S × T, T × S and T × T cases. By contrast, the sequential dependency in Figure 9.3 requires writing separate routines for these cases.

§9.2.1. Since `bamboos` is motivated to improve the previous discrete algorithms in Chapter 8, it is useful to first examine if there is any savings in FLOP count. §9.2.2 will restate the FLOP count for the old discrete algorithms, and comparison will be made in §9.2.3.

9.2.1 FLOP count for `bamboos` algorithms

Assuming $\boldsymbol{A_s}$ and $\boldsymbol{B_t}$ are two singletons, the following formally summarizes FLOP count for matrix cross-product of S × S type.

- For diagonal $\boldsymbol{W}$ (see algorithms in Figure 9.1a):
  - $\boldsymbol{A_s'WA_s}$: $\alpha_1 = O(n) + O(m_{\bar{A}_s} p^2_{\bar{A}_s})$;
  - $\boldsymbol{A_s'WB_t}$: $\alpha_2 = O(n) + O(K^{[1]}_{st} p_{\bar{B}_t}) + O(m_{\bar{A}_s} p_{\bar{A}_s} p_{\bar{B}_t})$ for right-to-left computation and $\alpha_3 = O(n) + O(K^{[1]}_{st} p_{\bar{A}_s}) + O(m_{\bar{B}_t} p_{\bar{A}_s} p_{\bar{B}_t})$ for left-to-right computation, where $K^{[1]}_{st}$ is the number of non-zero elements in $\bar{\boldsymbol{W}}$, which is upper bounded by $\min\{m_{\bar{A}_s} m_{\bar{B}_t},\ n\}$.
- For tri-diagonal $\boldsymbol{W}$ (see algorithm Figure 9.1b; note the $3n$ (or precisely $(3n-2)$) as there are three diagonals):
  - $\boldsymbol{A_s'WA_s}$: $\alpha_4 = O(n) + O(K^{[3]}_{ss} p_{\bar{A}_s}) + O(m_{\bar{A}_s} p^2_{\bar{A}_s})$, where $K^{[3]}_{ss}$ is the number of non-zero elements in $\bar{\boldsymbol{W}}$, which is upper bounded by $\min\{m^2_{\bar{A}_s},\ 3n\}$;
  - $\boldsymbol{A_s'WB_t}$: $\alpha_5 = O(n) + O(K^{[3]}_{st} p_{\bar{B}_t}) + O(m_{\bar{A}_s} p_{\bar{A}_s} p_{\bar{B}_t})$ for right-to-left computation and $\alpha_6 = O(n) + O(K^{[3]}_{st} p_{\bar{A}_s}) + O(m_{\bar{B}_t} p_{\bar{A}_s} p_{\bar{B}_t})$ for left-to-right computation, where $K^{[3]}_{st}$ is the number of non-zero elements in $\bar{\boldsymbol{W}}$, which is upper bounded by $\min\{m_{\bar{A}_s} m_{\bar{B}_t},\ 3n\}$.

Note that

- K is indexed by s and t to imply its dependence on $\boldsymbol{k}_{\bar{A}_s}$ and $\boldsymbol{k}_{\bar{B}_t}$. The superscript implies its dependence on the structure of $\boldsymbol{W}$: "[1]" for diagonal $\boldsymbol{W}$ and "[3]" for tri-diagonal $\boldsymbol{W}$.
- A run-time choice between left-to-right computation and right-to-left computation can be made, so for example, the realistic FLOP count for $\boldsymbol{A_s'WB_t}$ with diagonal $\boldsymbol{W}$ is $\min\{\alpha_2,\ \alpha_3\}$.

The unified representation in Figure 9.1.3 (or algorithm in Figure 9.4) makes it straightforward to derive FLOP count for S × T, T × S and T × T blocks; see Table 9.1. Note that

- The computational complexity for weights aggregation in α_1, α_2, ..., α_6 needs be adjusted to $O(stn)$ to account for the computation of $\boldsymbol{a}$ and $\boldsymbol{b}$ in Figure 9.4.
- I have replaced the big "O" notation with exact FLOP count. The constant proportionality for weights aggregation is 1 for diagonal $\boldsymbol{W}$ and 3 for tri-diagonal $\boldsymbol{W}$, while the constant proportionality for any matrix-matrix multiplication is 2.

Table 9.1: FLOP count for bamboos algorithms. The unified representation for $\boldsymbol{X_i}$ and $\boldsymbol{X_j}$ is used (see §9.1.3): $\boldsymbol{X_i}$ is a tensor with s packed margins from $(\bar{\boldsymbol{A}}_{\boldsymbol{1}},\ \boldsymbol{k}_{\bar{A}_1},\ m_{\bar{A}_1},\ p_{\bar{A}_1})$ to $(\bar{\boldsymbol{A}}_{\boldsymbol{s}},\ \boldsymbol{k}_{\bar{A}_s},\ m_{\bar{A}_s},\ p_{\bar{A}_s})$, and $\boldsymbol{X_j}$ is a tensor with t packed margins from $(\bar{\boldsymbol{B}}_{\boldsymbol{1}},\ \boldsymbol{k}_{\bar{B}_1},\ m_{\bar{B}_1},\ p_{\bar{B}_1})$ to $(\bar{\boldsymbol{B}}_{\boldsymbol{t}},\ \boldsymbol{k}_{\bar{B}_t},\ m_{\bar{B}_t},\ p_{\bar{B}_t})$. It is legitimate to have $s = 1$ and / or $t = 1$ so that a tensor degenerates to a singleton, in which case there is $p_{\dot{A}} = 1$ and / or $p_{\dot{B}} = 1$. Exact FLOP count instead of big "O" notation is given; see main text for more information.

$$
\begin{aligned}
\alpha_1 &= stn + 2m_{\bar{A}_s}p_{\bar{A}_s}^2 \\
\alpha_2 &= stn + 2K_{st}^{[1]}p_{\bar{B}_t} + 2m_{\bar{A}_s}p_{\bar{A}_s}p_{\bar{B}_t}, \quad K_{st}^{[1]} \leq \min\{m_{\bar{A}_s}m_{\bar{B}_t},\ n\} \\
\alpha_3 &= stn + 2K_{st}^{[1]}p_{\bar{A}_s} + 2m_{\bar{B}_t}p_{\bar{A}_s}p_{\bar{B}_t}, \quad K_{st}^{[1]} \leq \min\{m_{\bar{A}_s}m_{\bar{B}_t},\ n\} \\
\alpha_4 &= 3stn + 2K_{ss}^{[3]}p_{\bar{A}_s} + 2m_{\bar{A}_s}p_{\bar{A}_s}^2, \quad K_{ss}^{[3]} \leq \min\{m_{\bar{A}_s}^2,\ 3n\} \\
\alpha_5 &= 3stn + 2K_{st}^{[3]}p_{\bar{B}_t} + 2m_{\bar{A}_s}p_{\bar{A}_s}p_{\bar{B}_t}, \quad K_{st}^{[3]} \leq \min\{m_{\bar{A}_s}m_{\bar{B}_t},\ 3n\} \\
\alpha_6 &= 3stn + 2K_{st}^{[3]}p_{\bar{A}_s} + 2m_{\bar{B}_t}p_{\bar{A}_s}p_{\bar{B}_t}, \quad K_{st}^{[3]} \leq \min\{m_{\bar{A}_s}m_{\bar{B}_t},\ 3n\}
\end{aligned}
$$

	$\boldsymbol{X_i'WX_i} = (\dot{\boldsymbol{A}} \widetilde{\otimes} \boldsymbol{A_s})'\boldsymbol{W}(\dot{\boldsymbol{A}} \widetilde{\otimes} \boldsymbol{A_s})$	$\boldsymbol{X_i'WX_j} = (\dot{\boldsymbol{A}} \widetilde{\otimes} \boldsymbol{A_s})'\boldsymbol{W}(\dot{\boldsymbol{B}} \widetilde{\otimes} \boldsymbol{B_t})$
diagonal $\boldsymbol{W}$	$p_{\dot{A}}^2\alpha_1$	$p_{\dot{A}}p_{\dot{B}}\min\{\alpha_2,\ \alpha_3\}$
tri-diagonal $\boldsymbol{W}$	$p_{\dot{A}}^2\alpha_4$	$p_{\dot{A}}p_{\dot{B}}\min\{\alpha_5,\ \alpha_6\}$

9.2.2 A recap of FLOP count for the previous discrete algorithms

FLOP count for the old discrete algorithms was provided in captions of Figure 8.5a and Figure 8.5b, and the following is a recap.

- $\boldsymbol{A_s'WB_t}$: $O(np_{\bar{B}_t}) + O(m_{\bar{A}_s}p_{\bar{A}_s}p_{\bar{B}_t})$;
- $(\dot{\boldsymbol{A}} \widetilde{\otimes} \boldsymbol{A_s})'\boldsymbol{WB_t}$: $O(np_{\bar{B}_t}) + O(snp_{\dot{A}}p_{\bar{B}_t}/b) + O(np_{\dot{A}}p_{\bar{B}_t}) + O(m_{\bar{A}_s}p_{\bar{A}_s}p_{\bar{B}_t})$;
- $\boldsymbol{A_s'W}(\dot{\boldsymbol{B}} \widetilde{\otimes} \boldsymbol{B_t})$: $O(ntp_{\bar{B}_t}) + O(m_{\bar{A}_s}p_{\dot{A}}p_{\bar{A}_s}p_{\bar{B}_t})$;
- $(\dot{\boldsymbol{A}} \widetilde{\otimes} \boldsymbol{A_s})'\boldsymbol{W}(\dot{\boldsymbol{B}} \widetilde{\otimes} \boldsymbol{B_t})$: $O(ntp_{\dot{B}}p_{\bar{B}_t}) + O(snp_{\dot{A}}p_{\dot{B}}p_{\bar{B}_t}/b) + O(np_{\dot{A}}p_{\dot{B}}p_{\bar{B}_t}) + O(m_{\bar{A}_s}p_{\dot{A}}p_{\bar{A}_s}p_{\dot{B}}p_{\bar{B}_t})$.

If the column chunk size b is set to $20s$ as is discussed in §8.4, the $O(snp_{\dot{A}}p_{\dot{B}}p_{\bar{B}_t}/b)$ for example, will be negligible compared with $O(np_{\dot{A}}p_{\dot{B}}p_{\bar{B}_t})$. The unified representation between singleton and tensor can then be applied to simplify the results for all 4 cases to $O(ntp_{\dot{B}}p_{\bar{B}_t}) + O(np_{\dot{A}}p_{\dot{B}}p_{\bar{B}_t}) + O(m_{\bar{A}_s}p_{\dot{A}}p_{\bar{A}_s}p_{\dot{B}}p_{\bar{B}_t})$. To ease comparison with FLOP counts for `bamboos` algorithms (see Table 9.1), it is helpful to replace the big "O" notation with exact FLOP count. This proceeds as follows.

- The $O(ntp_{\dot{B}}p_{\bar{B}_t})$ complexity is associated with unpacking and reweighting of $\dot{\boldsymbol{B}} \widetilde{\otimes} \boldsymbol{B_t}$. For the former, the constant proportionality is 1; for the latter, the constant proportionality is 1 if $\boldsymbol{W}$ is diagonal and 6 if $\boldsymbol{W}$ is tri-diagonal.
- The $O(np_{\dot{A}}p_{\dot{B}}p_{\bar{B}_t})$ complexity is for matrix row-aggregation. The constant proportionality is 1.
- The $O(m_{\bar{A}_s}p_{\dot{A}}p_{\bar{A}_s}p_{\dot{B}}p_{\bar{B}_t})$ complexity is for matrix cross-product, so constant proportionality is 2.

To summarize, the FLOP count for computing $(\dot{\boldsymbol{A}} \widetilde{\otimes} \boldsymbol{A_s})'\boldsymbol{W}(\dot{\boldsymbol{B}} \widetilde{\otimes} \boldsymbol{B_t})$ is

- $2ntp_{\dot{B}}p_{\bar{B}_t} + np_{\dot{A}}p_{\dot{B}}p_{\bar{B}_t} + 2m_{\bar{A}_s}p_{\dot{A}}p_{\bar{A}_s}p_{\dot{B}}p_{\bar{B}_t}$ for diagonal $\boldsymbol{W}$;
- $7ntp_{\dot{B}}p_{\bar{B}_t} + np_{\dot{A}}p_{\dot{B}}p_{\bar{B}_t} + 2m_{\bar{A}_s}p_{\dot{A}}p_{\bar{A}_s}p_{\dot{B}}p_{\bar{B}_t}$ for tri-diagonal $\boldsymbol{W}$;

Still, recall that $(\dot{\boldsymbol{B}} \widetilde{\otimes} \boldsymbol{B_t})'\boldsymbol{W}(\dot{\boldsymbol{A}} \widetilde{\otimes} \bar{\boldsymbol{A}}_{\boldsymbol{s}})$ may be computed instead of $(\dot{\boldsymbol{A}} \widetilde{\otimes} \boldsymbol{A_s})'\boldsymbol{W}(\dot{\boldsymbol{B}} \widetilde{\otimes} \boldsymbol{B_t})$ if it is cheaper. The FLOP count for this transposed variant is

Table 9.2: FLOP count for previous discrete algorithms in Chapter 8. Raw results are rearranged to the following form to facilitate comparison with Table 9.1; see main text for details of derivation.

$$
\begin{aligned}
\gamma_1 &= np_{\bar{A}_s}(1 + 2s/p_{\dot{A}}) + 2m_{\bar{A}_s}p^2_{\bar{A}_s} \\
\gamma_2 &= np_{\bar{B}_t}(1 + 2t/p_{\dot{A}}) + 2m_{\bar{A}_s}p_{\bar{A}_s}p_{\bar{B}_t} \\
\gamma_3 &= np_{\bar{A}_s}(1 + 2s/p_{\dot{B}}) + 2m_{\bar{B}_t}p_{\bar{A}_s}p_{\bar{B}_t} \\
\gamma_4 &= np_{\bar{A}_s}(1 + 7s/p_{\dot{A}}) + 2m_{\bar{A}_s}p^2_{\bar{A}_s} \\
\gamma_5 &= np_{\bar{B}_t}(1 + 7t/p_{\dot{A}}) + 2m_{\bar{A}_s}p_{\bar{A}_s}p_{\bar{B}_t} \\
\gamma_6 &= np_{\bar{A}_s}(1 + 7s/p_{\dot{B}}) + 2m_{\bar{B}_t}p_{\bar{A}_s}p_{\bar{B}_t}
\end{aligned}
$$

	$\boldsymbol{X_i'WX_i} = (\dot{\boldsymbol{A}} \widetilde{\otimes} \boldsymbol{A_s})'\boldsymbol{W}(\dot{\boldsymbol{A}} \widetilde{\otimes} \boldsymbol{A_s})$	$\boldsymbol{X_i'WX_j} = (\dot{\boldsymbol{A}} \widetilde{\otimes} \boldsymbol{A_s})'\boldsymbol{W}(\dot{\boldsymbol{B}} \widetilde{\otimes} \boldsymbol{B_t})$
diagonal $\boldsymbol{W}$	$p^2_{\dot{A}}\gamma_1$	$p_{\dot{A}}p_{\dot{B}}\min\{\gamma_2,\ \gamma_3\}$
tri-diagonal $\boldsymbol{W}$	$p^2_{\dot{A}}\gamma_4$	$p_{\dot{A}}p_{\dot{B}}\min\{\gamma_5,\ \gamma_6\}$

- $2nsp_{\dot{A}}p_{\bar{A}_s} + np_{\dot{A}}p_{\dot{B}}p_{\bar{A}_s} + 2m_{\bar{B}_t}p_{\dot{B}}p_{\bar{B}_t}p_{\dot{A}}p_{\bar{A}_s}$ for diagonal $\boldsymbol{W}$;
- $7nsp_{\dot{A}}p_{\bar{A}_s} + np_{\dot{A}}p_{\dot{B}}p_{\bar{A}_s} + 2m_{\bar{B}_t}p_{\dot{B}}p_{\bar{B}_t}p_{\dot{A}}p_{\bar{A}_s}$ for tri-diagonal $\boldsymbol{W}$;

To facilitate comparison with `bamboos`, it helps to rearrange above results in a similar format to Table 9.1. This only involves a little more paper work, and Table 9.2 presents such results. In this way, it is sufficient to compare α_i with γ_i to draw conclusions.

9.2.3 FLOP count comparison and summary

It is not easy to compare FLOP count of `bamboos` algorithms and the old discrete algorithms. A message from Table 9.1 is that realistic complexity of `bamboos` algorithms is data dependent via those K values. It is very crude to replace K values by their upper bound, but this at least provides some idea / guidance.

In the best case where $m_{\bar{A}_s}m_{\bar{B}_t} < n$ or $m_{\bar{A}_s}m_{\bar{B}_t} < 3n$, there are

$$
\begin{aligned}
\alpha_1 &= stn + 2m_{\bar{A}_s}p^2_{\bar{A}_s}, \\
\alpha_2 &\le stn + 2m_{\bar{A}_s}m_{\bar{B}_t}p_{\bar{B}_t} + 2m_{\bar{A}_s}p_{\bar{A}_s}p_{\bar{B}_t}, \\
\alpha_3 &\le stn + 2m_{\bar{A}_s}m_{\bar{B}_t}p_{\bar{A}_s} + 2m_{\bar{B}_t}p_{\bar{A}_s}p_{\bar{B}_t}, \\
\alpha_4 &\le 3stn + 2m^2_{\bar{A}_s}p_{\bar{A}_s} + 2m_{\bar{A}_s}p^2_{\bar{A}_s}, \\
\alpha_5 &\le 3stn + 2m_{\bar{A}_s}m_{\bar{B}_t}p_{\bar{B}_t} + 2m_{\bar{A}_s}p_{\bar{A}_s}p_{\bar{B}_t}, \\
\alpha_6 &\le 3stn + 2m_{\bar{A}_s}m_{\bar{B}_t}p_{\bar{A}_s} + 2m_{\bar{B}_t}p_{\bar{A}_s}p_{\bar{B}_t},
\end{aligned}
$$

so that the only obviously n dependent terms are associated with weights aggregation. In large GAM fitting, it is common that 1) $n \to \infty$, 2) $m_{\bar{A}_s}$ and $m_{\bar{B}_t}$ grows sublinearly with n and 3) $p_{\bar{A}_s}$ and $p_{\bar{B}_t}$ grows sublinearly with $m_{\bar{A}_s}$ and $m_{\bar{B}_t}$. Note that since there is $m_{\bar{A}_s}m_{\bar{B}_t} < n$ or $m_{\bar{A}_s}m_{\bar{B}_t} < 3n$, the growth rate of $m_{\bar{A}_s}$ and $m_{\bar{B}_t}$ is at most $n^{\frac{1}{2}}$. Let us consider two scenarios.

- $n \to \infty$ and the growth rate of $m_{\bar{A}_s}$ and $m_{\bar{B}_t}$ is smaller than $n^{\frac{1}{2}}$. In this case, complexity in weights aggregation is dominant. Asymptotically there are $\alpha_1 \approx \alpha_2 \approx \alpha_3 \approx stn$, $\alpha_4 \approx \alpha_5 \approx \alpha_6 \approx 3stn$, $\gamma_1 \approx \gamma_3 \approx \gamma_4 \approx \gamma_6 \approx np_{\bar{A}_s}$ and $\gamma_2 \approx \gamma_5 \approx np_{\bar{B}_t}$. `bamboos` algorithms yield huge computational savings.
- $n \to \infty$ and the growth rate of $m_{\bar{A}_s}$ and $m_{\bar{B}_t}$ attains $n^{\frac{1}{2}}$. In this case, complexity in matrix-matrix multiplication accounts for a fixed proportion of gross complexity. It may also be non-negligible compared with weights aggregation. Analysis based on upper bound gives no useful conclusion, because for example, if $m_{\bar{A}_s}m_{\bar{B}_t}$ is almost n for diagonal $\boldsymbol{W}$, the upper bound of α_3 is almost

$2np_{\bar{A}_s}$, readily larger than $\gamma_3 \approx np_{\bar{A}_s}$. If $m_{\bar{A}_s}m_{\bar{B}_t}$ is almost $3n$ for tri-diagonal $\boldsymbol{W}$, the upper bound of α_6 is almost $6np_{\bar{A}_s}$, even larger than $\gamma_6 \approx np_{\bar{A}_s}$. However, if $m_{\bar{A}_s}m_{\bar{B}_t}$ is really this large, the real situation is likely to be that $\bar{\boldsymbol{W}}$ is very sparse, therefore the upper bound used here is too much bigger than K values that actually determine the computational complexity.

For the worst case where $m_{\bar{A}_s}m_{\bar{B}_t} \geq n$ or $m_{\bar{A}_s}m_{\bar{B}_t} \geq 3n$, the situation is basically as same as the second limiting case above. No precise comparison can be made when the sparsity of $\bar{\boldsymbol{W}}$ steps in. However, this does not imply that `bamboos` has no advantage. In fact, the advantage may be magnificent. A sparse $\bar{\boldsymbol{W}}$ can mean that $K \approx 0.1n$, $K \approx 0.01n$ or even $K \approx 0.001n$, therefore the first limiting case above is asymptotically recovered.

To some degree, the superiority of `bamboos`, if any, can only be established on practical benchmarking. This will be done later in §9.4. However, at least one firm conclusion can be made from the analysis so far: it is important to make weights aggregation as fast as possible, as that is what `bamboos` algorithms will be primarily doing in the $n \to \infty$ limiting case. In the next section, I will demonstrate how to maximize the performance of weights aggregation. In addition, if you have read previous sections carefully, you may have already realized that although I have mentioned sparsity many times by now, nothing is really said about how to detect whether $\bar{\boldsymbol{W}}$ is dense or sparse, and if sparse, how to represent it. The reason is that the "dense-sparse switch" for $\bar{\boldsymbol{W}}$ construction as well as the sparse representation of $\bar{\boldsymbol{W}}$ is embedded in the performance maximization of weights aggregation in `bamboos` implementation. This will be clear in the next section.

9.3 Fast weights aggregation for $\bar{W}$ and its sparse construction

The way to maximize the performance of weights aggregation is subtle. Since it has been established that computing $\boldsymbol{X}_{\boldsymbol{i}}'\boldsymbol{W}\boldsymbol{X}_{\boldsymbol{j}}$ of S × S type plays a pivotal role in `bamboos` (see Figure 9.5), it might appear that performance optimization of weights aggregation in computing S × S blocks is the way to go and any gains will automatically be broadcasted to computations of other types of blocks. However, this is not true. In fact, there is no way to enhance the performance of weights aggregation in S × S case; optimization opportunity is found instead in the repeated application of the same kind of weights aggregation in S × T, T × S and T × T blocks. In the rest of this section,

- §9.3.1 introduces the idea of nested sorting, run-length encoding and bin aggregation that makes weights aggregation as fast as possible;
- §9.3.2 explains the sorting algorithms used in the nested sorting;
- §9.3.3 demonstrates how nested sorting, run-length encoding and bin aggregation facilitates sparse representation of $\bar{\boldsymbol{W}}$, and how the "dense-sparse switch" for $\bar{\boldsymbol{W}}$ works in `bamboos`.

Note that for clarity, all these sections are discussed using diagonal $\boldsymbol{W}$. How tri-diagonal $\boldsymbol{W}$ is dealt with will be briefly mentioned in §9.3.4 in the end.

9.3.1 Nested sorting, run-length encoding and bin aggregation

Consider constructing $\bar{\boldsymbol{W}}$ as a dense matrix when computing an S × S block $\boldsymbol{A}_{\boldsymbol{s}}'\boldsymbol{W}\boldsymbol{B}_{\boldsymbol{t}}$ where $\boldsymbol{W}$ is diagonal. Algorithm in the right panel of Figure 9.1a hints that elements of $\bar{\boldsymbol{W}}$ are likely to be accessed randomly over the entire matrix during the , as entries of index vectors $\boldsymbol{k}_{\bar{A}}$ and especially $\boldsymbol{k}_{\bar{B}}$ are not sorted. Therefore, the $O(n)$ random memory access behind the $O(n)$ FLOP in aggregation

original pair	primary sort		secondary sort	
$(\boldsymbol{k}_{\bar{A}_s},\ \boldsymbol{k}_{\bar{B}_t})$	$(\boldsymbol{k}_{\bar{A}_s}^{[1]},\ \boldsymbol{k}_{\bar{B}_t}^{[1]})$	$\boldsymbol{\sigma}^{[1]}$	$(\boldsymbol{k}_{\bar{A}_s}^{[2]},\ \boldsymbol{k}_{\bar{B}_t}^{[2]})$	$\boldsymbol{\sigma}^{[2]}$
(1, 4)	(2, 1)	5	(2, 1)	5
(4, 4)	(3, 1)	9	(2, 1)	12
(1, 3)	(2, 1)	12	(3, 1)	9
(4, 3)	(1, 2)	10	(1, 2)	10
(2, 1)	(2, 2)	11	(2, 2)	11
(4, 3)	(1, 3)	3	(1, 3)	3
(1, 4)	(4, 3)	4	(1, 3)	16
(3, 3)	(4, 3)	6	(3, 3)	8
(3, 1)	(3, 3)	8	(4, 3)	4
(1, 2)	(1, 3)	16	(4, 3)	6
(2, 2)	(1, 4)	1	(1, 4)	1
(2, 1)	(4, 4)	2	(1, 4)	7
(4, 4)	(1, 4)	7	(1, 4)	15
(4, 4)	(4, 4)	13	(4, 4)	2
(1, 4)	(4, 4)	14	(4, 4)	13
(1, 3)	(1, 4)	15	(4, 4)	14

Figure 9.6: An illustration of the nested sorting on $(\boldsymbol{k}_{\bar{A}_s},\ \boldsymbol{k}_{\bar{B}_t})$ pairs, using an example with $m_{\bar{A}_s} = m_{\bar{B}_t} = 4$ and $n = 16$. Primary sort arranges $\boldsymbol{k}_{\bar{B}_t}$ into $\boldsymbol{k}_{\bar{B}_t}^{[1]}$ whose entries are in non-descending order, while generating a permutation index vector $\boldsymbol{\sigma}^{[1]}$. $\boldsymbol{k}_{\bar{A}_s}$ is permutated accordingly to $\boldsymbol{k}_{\bar{A}_s}^{[1]}$. At this stage $\boldsymbol{k}_{\bar{B}_t}^{[1]}$ can be cut into several "bins", i.e., segments with runs of the value. Secondary sort is a bin-wise sort. Entries of $\boldsymbol{k}_{\bar{A}_s}^{[1]}$ in each bin are sorted in non-descending order to $\boldsymbol{k}_{\bar{A}_s}^{[2]}$ and $\boldsymbol{\sigma}^{[1]}$ is permutated accordingly to $\boldsymbol{\sigma}^{[2]}$. Note that this stage has no effect on $\boldsymbol{k}_{\bar{B}_t}^{[1]}$ so $\boldsymbol{k}_{\bar{B}_t}^{[2]} = \boldsymbol{k}_{\bar{B}_t}^{[1]}$. In the end of the nested sorting, $(\boldsymbol{k}_{\bar{A}_s}^{[2]},\ \boldsymbol{k}_{\bar{B}_t}^{[2]})$ admits a bin structure.

is generally much more expensive, when $\bar{\boldsymbol{W}}$ is not small enough to fit in CPU cache. For example, a 100×100 matrix of double-precision floating-point numbers is already 150% larger than a 32 KB L1 data cache.

Such inefficiency can be tolerated in computing the above S $\times$ S block. But consider, for example the computation of a T $\times$ T block $(\dot{\boldsymbol{A}} \widetilde{\otimes} \boldsymbol{A_s})'\boldsymbol{W}(\dot{\boldsymbol{B}} \widetilde{\otimes} \boldsymbol{B_t})$ for diagonal $\boldsymbol{W}$, the same type of weights aggregation is repeatedly performed $p_{\dot{A}}p_{\dot{B}}$ times through the iteration computation of $\boldsymbol{A_s'W_{gh}B_t}$ (see Figure 9.4). By "the same type" I mean that the access pattern to $\bar{\boldsymbol{W}}_{\boldsymbol{gh}}$ is exactly the same for any g and h; only values of $\boldsymbol{W_{gh}}$ and $\bar{\boldsymbol{W}}_{\boldsymbol{gh}}$ are changing from iteration to iteration. If this common access pattern can be "memorized" somehow, i.e., the loopup to $\boldsymbol{k}_{\bar{A}_s}$ and $\boldsymbol{k}_{\bar{B}_t}$ is only done once rather than $p_{\dot{A}}p_{\dot{B}}$ times, the efficiency of weights aggregation would be much improved.

One way to do this is by a nested sorting on $(\boldsymbol{k}_{\bar{A}_s},\ \boldsymbol{k}_{\bar{B}_t})$, which includes the following two stages.

- A *primary sort* $\boldsymbol{\sigma}^{[1]} : (\boldsymbol{k}_{\bar{A}_s},\ \boldsymbol{k}_{\bar{B}_t}) \to (\boldsymbol{k}_{\bar{A}_s}^{[1]},\ \boldsymbol{k}_{\bar{B}_t}^{[1]})$, that sorts $\boldsymbol{k}_{\bar{B}_t}$ in non-descending order to $\boldsymbol{k}_{\bar{B}_t}^{[1]}$. Note that $\boldsymbol{k}_{\bar{A}_s}$ needs be permutated according to the permutation index vector $\boldsymbol{\sigma}^{[1]}$ to $\boldsymbol{k}_{\bar{A}_s}^{[1]}$, so that $(\boldsymbol{k}_{\bar{A}_s}^{[1]},\ \boldsymbol{k}_{\bar{B}_t}^{[1]})$ and $(\boldsymbol{k}_{\bar{A}_s},\ \boldsymbol{k}_{\bar{B}_t})$ have the same pairwise relationship.
- A *secondary sort* $\boldsymbol{\sigma}^{*} : (\boldsymbol{k}_{\bar{A}_s}^{[1]},\ \boldsymbol{k}_{\bar{B}_t}^{[1]},\ \boldsymbol{\sigma}^{[1]}) \to (\boldsymbol{k}_{\bar{A}_s}^{[2]},\ \boldsymbol{k}_{\bar{B}_t}^{[2]},\ \boldsymbol{\sigma}^{[2]})$, that further sorts entries of $\boldsymbol{k}_{\bar{A}_s}^{[1]}$ in non-descending order into $\boldsymbol{k}_{\bar{A}_s}^{[2]}$ for each "bin" (i.e., segment of the same value) of the sorted $\boldsymbol{k}_{\bar{B}_t}^{[1]}$. The permutation index $\boldsymbol{\sigma}^{*}$ is not of interest; rather, $\boldsymbol{\sigma}^{[2]}$ should be recorded. Note that this bin-wise sorting has no effect on $\boldsymbol{k}_{\bar{B}_t}^{[1]}$ so $\boldsymbol{k}_{\bar{B}_t}^{[2]} = \boldsymbol{k}_{\bar{B}_t}^{[1]}$.

At the end of the nested sorting, $(\boldsymbol{k}_{\bar{A}_s}^{[2]},\ \boldsymbol{k}_{\bar{B}_t}^{[2]})$ will be partitioned into "bins". This ensures that during weights aggregation, elements of $\bar{\boldsymbol{W}}$ will be accessed sequentially. See Figure 9.6 for an illustration.

The nested sorting changes the way that $\bar{\boldsymbol{W}}_{\boldsymbol{gh}}$ is computed. The resulting permutation $\boldsymbol{\sigma}^{[2]}$ must be applied to $\boldsymbol{W_{gh}} = \texttt{diag}(\boldsymbol{w_{gh}})$. An implication is that while elements of $\bar{\boldsymbol{W}}_{\boldsymbol{gh}}$ are accessed sequentially, elements of $\boldsymbol{w_{gh}}$ are now accessed in a random fashion. This reverses the situation before nested

sorting where elements of $\bar{\boldsymbol{W}}_{\boldsymbol{gh}}$ are accessed randomly and elements of $\boldsymbol{w}_{\boldsymbol{gh}}$ are accessed sequentially. No practical gains will be attained in this way. Fortunately, there is a workaround. Note that the permutation of $\boldsymbol{w}_{\boldsymbol{gh}}$ is

$$\dot{\boldsymbol{w}}_{\boldsymbol{gh}} := \boldsymbol{w}_{\boldsymbol{gh}}(\boldsymbol{\sigma}^{[2]}) = \texttt{diag}(\dot{\boldsymbol{A}}(\boldsymbol{\sigma}^{[2]},\ g))\texttt{diag}(\boldsymbol{w}(\boldsymbol{\sigma}^{[2]}))\texttt{diag}(\dot{\boldsymbol{B}}(\boldsymbol{\sigma}^{[2]},\ h)) = \dot{\boldsymbol{A}}(\boldsymbol{\sigma}^{[2]},\ g)\circ\boldsymbol{w}(\boldsymbol{\sigma}^{[2]})\circ\dot{\boldsymbol{B}}(\boldsymbol{\sigma}^{[2]},\ h),$$

where "$\circ$" is the Hadamard product between two vectors (see the footnote in §8.3 if you forget what it is). The following observations and solutions can then be made.

- $\boldsymbol{w}(\boldsymbol{\sigma}^{[2]})$ is loop-invariant w.r.t. g and h so it can be pre-computed.
- Computation of $\dot{\boldsymbol{A}}(\boldsymbol{\sigma}^{[2]},\ g)$ and $\dot{\boldsymbol{B}}(\boldsymbol{\sigma}^{[2]},\ h)$ is basically no different to computation of $\dot{\boldsymbol{A}}(,\ g)$ and $\dot{\boldsymbol{B}}(,\ h)$. Algorithm in Figure 8.3c still applies, by now using $\boldsymbol{k}_{\bar{\boldsymbol{A}}_{\boldsymbol{u}}}(\boldsymbol{\sigma}^{[2]})$, $u = 1,\ 2,\ ...,\ (s-1)$ and $\boldsymbol{k}_{\bar{\boldsymbol{B}}_{\boldsymbol{v}}}(\boldsymbol{\sigma}^{[2]})$, $v = 1,\ 2,\ ...,\ (v-1)$ as index vectors when unpacking columns of $\bar{\boldsymbol{A}}_{\boldsymbol{u}}$ and $\bar{\boldsymbol{B}}_{\boldsymbol{v}}$. All these index vectors are loop-invariant hence can be pre-computed.

In this way, the negative effect of permutation is eliminated in the double loop nest. In practice, pre-permutation of $\boldsymbol{w}$, $\boldsymbol{k}_{\bar{\boldsymbol{A}}_{\boldsymbol{u}}}$, $u = 1,\ 2,\ ...,\ (s-1)$ and $\boldsymbol{k}_{\bar{\boldsymbol{B}}_{\boldsymbol{v}}}$, $v = 1,\ 2,\ ...,\ (v-1)$ can be implemented in secondary sort as soon as $\boldsymbol{\sigma}^{[\boldsymbol{2}]}$ is obtained.

What the nestes sorting achieves in practice, is that it casts the aggregation of $\boldsymbol{w}_{\boldsymbol{gh}}$ over $(\boldsymbol{k}_{\bar{\boldsymbol{A}}_s},\ \boldsymbol{k}_{\bar{\boldsymbol{B}}_t})$ to an equivalent aggregation of $\dot{\boldsymbol{w}}_{\boldsymbol{gh}}$ over $(\boldsymbol{k}^{[2]}_{\bar{\boldsymbol{A}}_s},\ \boldsymbol{k}^{[2]}_{\bar{\boldsymbol{B}}_t})$. While cache-unfriendly random memory access is a performance penalty for the former, sequential memory access is almost guaranteed for latter. I used "almost" because unpacking columns of $\bar{\boldsymbol{A}}_{\boldsymbol{u}}$ and $\bar{\boldsymbol{B}}_{\boldsymbol{v}}$ still involves random memory access. But this is what has to be paid for using packed storage and it exists in both aggregation problems. Note that since $(\boldsymbol{k}^{[2]}_{\bar{\boldsymbol{A}}_s},\ \boldsymbol{k}^{[2]}_{\bar{\boldsymbol{B}}_t})$ admits "bin" structures, aggregation of $\dot{\boldsymbol{w}}_{\boldsymbol{gh}}$ is a bin-wise aggregation. To facilitate such aggregation, perform a *run-length enconding* for $(\boldsymbol{k}^{[2]}_{\bar{\boldsymbol{A}}_s},\ \boldsymbol{k}^{[2]}_{\bar{\boldsymbol{B}}_t})$, producing the following triplets:

- $\boldsymbol{i} = (2,\ 3,\ 1,\ 2,\ 1,\ 3,\ 4,\ 1,\ 4)$;
- $\boldsymbol{j} = (1,\ 1,\ 2,\ 2,\ 3,\ 3,\ 3,\ 4,\ 4)$;
- $\boldsymbol{f} = (2,\ 1,\ 1,\ 1,\ 2,\ 1,\ 2,\ 3,\ 3)$,

which imply that $\bar{\boldsymbol{W}}(\boldsymbol{i}(v),\ \boldsymbol{j}(v))$, $v = 1,\ 2,\ \cdots,\ 9$ is the sum of $\boldsymbol{f}(v)$ consecutive elements in $\dot{\boldsymbol{w}}_{\boldsymbol{gh}}$. The number of elements in those triplets is the number of bins (here, 9). It is also $K^{[1]}_{st}$, the number of non-zero elements in $\bar{\boldsymbol{W}}$. Given these triplets, the bin aggregation can be implemented with the double loop nest in Figure 9.7.

Note that while this seems more complicated than the single aggregation loop based on $(\boldsymbol{k}_{\bar{\boldsymbol{A}}_s},\ \boldsymbol{k}_{\bar{\boldsymbol{B}}_t})$ and $\boldsymbol{w}_{\boldsymbol{gh}}$ (see the right panel of Figure 9.1a), it is in practice faster, because of the optimized memory access and the use of accumulator variable a to reduce the amount of write-back. Figure 9.8 benchmarks two aggregation methods on Intel Core i5-2557M. Note that for fixed n, as $\bar{\boldsymbol{W}}$ gets larger and larger, the simple aggregation method is increasingly time consuming, which reflects the performance penalty from random memory access. On the other hand, the operation time for bin aggregation almost stays constant. On a 141×141 $\bar{\boldsymbol{W}}$ (with about 20000 elements), the bin aggregation method is 3.9 times as fast.

9.3.2 Sorting algorithms in `bamboos`

In this section, I will explain the sorting algorithms used for primary sort and secondary sort. There are tons of sorting methods in general, so strictly speaking, careful choice should be made between

initialize $\bar{\boldsymbol{W}}$ as a dense matrix
$\bar{\boldsymbol{W}} = \text{zeros}(m_{\bar{\boldsymbol{A}}_s},\ m_{\bar{\boldsymbol{B}}_t})$
number of elements to be skipped to reach the first element in a bin
$s = 0$
loop through bins
for $v = 1 : K_{st}^{[1]}$
    number of values to be aggregated
    $f = \boldsymbol{f}(v)$
    an accumulator variable in a CPU register
    $a = 0$
    aggregation on the v^{th} bin
    for $u = 1 : f$
        $a \mathrel{+}= \dot{\boldsymbol{w}}_{\boldsymbol{gh}}(s + u)$
    write-back
    $\bar{\boldsymbol{W}}(\boldsymbol{i}(v),\ \boldsymbol{j}(v)) = a$
    $s \mathrel{+}= f$

Figure 9.7: Fast bin aggregation that produces a dense $\bar{\boldsymbol{W}}$. The triplets $\boldsymbol{i}$, $\boldsymbol{j}$ and $\boldsymbol{f}$ are the result of run-length encoding on $(\boldsymbol{k}_{\bar{\boldsymbol{A}}_s}^{[2]},\ \boldsymbol{k}_{\bar{\boldsymbol{B}}_t}^{[2]})$ (see main text for more information). Run-length encoding also reports the number of bins, which is in fact $K_{st}^{[1]}$, the number of non-zero elements in $\bar{\boldsymbol{W}}$. Note that talking about non-zero elements of $\bar{\boldsymbol{W}}$ does not necessarily mean treating the matrix as sparse. However, the nested sorting, run-length encoding and bin aggregation do make it straightforward to construct $\bar{\boldsymbol{W}}$ as sparse instead of dense. This will be clear in §9.3.3.

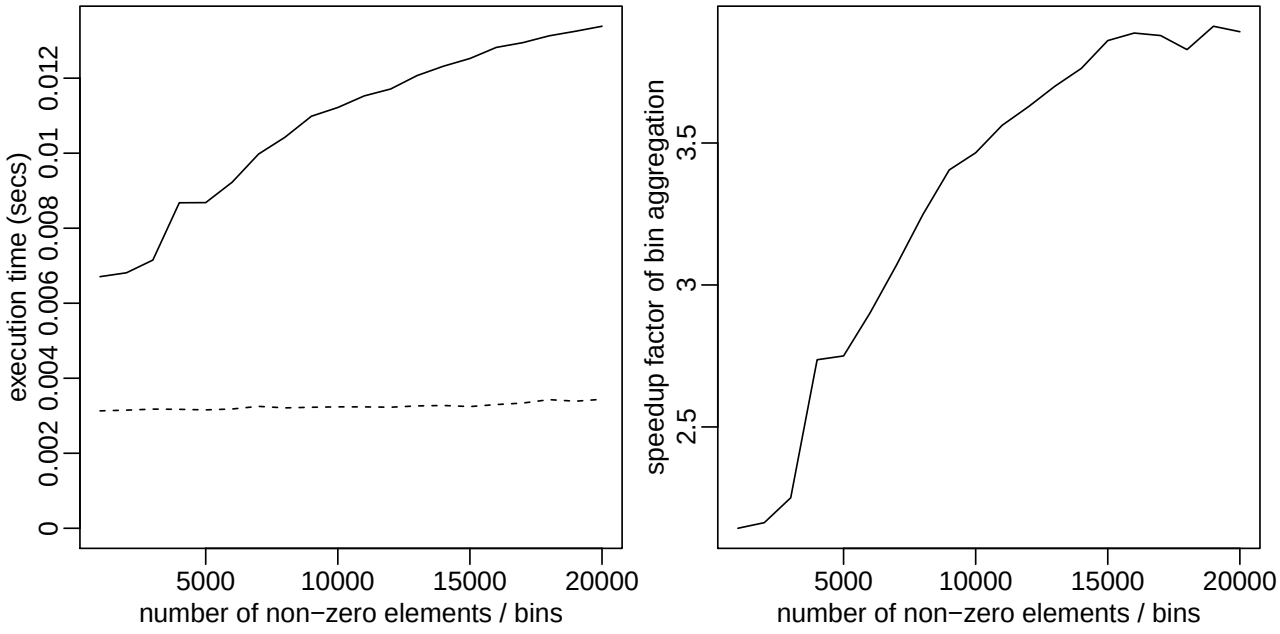


Figure 9.8: Benchmarking simple aggregation (see the *for* loop in the right panel of Figure 9.1a) and bin aggregation (see Figure 9.7) on Intel Core i5-2557M (see Appendix A for hardware information). A weight vector of length $n = 10^6$ is fixed for the experiment, and the size of $\bar{\boldsymbol{W}}$ grows. The left panel sketches the execution time (in seconds) for both methods (solid line for simple aggregation and dashed line for bin aggregation), and the right panel shows how faster bin aggregation is compared with simple aggregation. As $\bar{\boldsymbol{W}}$ gets larger and larger, the simple aggregation method is increasingly time consuming, reflecting the performance penalty from random memory access. On the other hand, the operation time for bin aggregation almost stays constant. On a 141×141 $\bar{\boldsymbol{W}}$ (with about 20000 elements), the bin aggregation method is 3.9 times as fast.

different methods. However, sorting is not a computational hotspot in `bamboos`, so there is no need to do premature optimization on this topic. Besides, sorting tasks in `bamboos` are integer sorting problems, and there are a few algorithms known to be best at such tasks. These include counting sort, radix sort (and maybe quicksort). At the moment, `bamboos` relies on counting sort for primary sort; this is guaranteed to be optimal. Binary MSD (most significant digit) radix sort is used for secondary sort; it may not be optimal but it is also good enough.

Counting sort algorithm for primary sort

Primary sort is an integer sorting problem, where a length-n vector $\boldsymbol{k}_{\bar{B}_t}$ taking values in $1,\ 2,\ \cdots,\ m_{\bar{B}_t}$ needs be sorted in ascending order. In particular, $n \gg m_{\bar{B}_t}$. Counting sort is optimal for this task, with $O(n)$ time complexity and $O(m_{\bar{B}_t})$ space complexity. In the end of the sorting, it returns the required permutation index $\boldsymbol{\sigma}^{[1]}$ and a frequency table $\boldsymbol{f}^{[1]}$ for $1,\ 2,\ \cdots,\ m$. Note that $\boldsymbol{k}_{\bar{B}_t}^{[\mathbf{1}]}$ is not returned, but $\boldsymbol{f}^{[1]}$ is equally informative on how $(\boldsymbol{k}_{\bar{A}_s}^{[\mathbf{1}]},\ \boldsymbol{k}_{\bar{B}_t}^{[\mathbf{1}]})$ is cut into bins according to $\boldsymbol{k}_{\bar{B}_t}^{[\mathbf{1}]}$.

The core idea of counting sort is as what its name suggests: counting. Consider the following length-20 toy integer vectors taking values 1, 2, ..., 7.

$$\boldsymbol{k}_{\bar{B}_t} = 3,\ 7,\ 2,\ 5,\ 1,\ 4,\ 6,\ 2,\ 5,\ 6,\ 1,\ 7,\ 3,\ 3,\ 4,\ 2,\ 1,\ 5,\ 6,\ 6$$

The first round of counting produces $\boldsymbol{f}^{[1]}$, and a vector of pointers $\boldsymbol{p}$:

	1	2	3	4	5	6	7
$\boldsymbol{f}^{[1]}$ =	3	3	3	2	3	4	2
$\boldsymbol{p}$ =	0	3	6	9	11	14	18

Here, a "header" is written in gray to help you associate values of $\boldsymbol{f}^{[1]}$ and $\boldsymbol{p}$ to unique integer values in $\boldsymbol{k}_{\bar{B}_t}$. If the goal of counting sort is just to obtain $\boldsymbol{k}_{\bar{B}_t}^{[\mathbf{1}]}$, then the sorting readily completes, because the following sorted array is simply generated by replicating 1, 2, ..., 7 according to the frequency table $\boldsymbol{f}^{[1]}$:

$$\boldsymbol{k}_{\bar{B}_t}^{[\mathbf{1}]} = |1,\ 1,\ 1,\ |2,\ 2,\ 2,\ |3,\ 3,\ 3,\ |4,\ 4,\ |5,\ 5,\ 5,\ |6,\ 6,\ 6,\ 6,\ |7,\ 7$$

This also makes it easy to explain what $\boldsymbol{p}$ is. It is just the pointer to those vertical bars: the number of elements before these bars.

When the goal of counting sort is to obtain $\boldsymbol{\sigma}^{[1]}$, a bit more work is needed. We have to scan $\boldsymbol{k}_{\bar{B}_t}$ for the second time to determine for example, which "1" in $\boldsymbol{k}_{\bar{B}_t}$ is the first "1" in $\boldsymbol{k}_{\bar{B}_t}^{[1]}$, which "1" in $\boldsymbol{k}_{\bar{B}_t}$ is the second "1" in $\boldsymbol{k}_{\bar{B}_t}^{[1]}$. This is essentially a second round of counting. To start with, a counter vector $\boldsymbol{c}$ is initialized:

	1	2	3	4	5	6	7
$\boldsymbol{c}$ =	0	0	0	0	0	0	0

Then the scanning of $\boldsymbol{k}_{\bar{B}_t}$ starts. The first value is $\boldsymbol{k}_{\bar{B}_t}(1) = 3$, so the counter $\boldsymbol{c}(3)$ is incremented to record that the first "3" in $\boldsymbol{k}_{\bar{B}_t}^{[1]}$ has been met. At the same time, it is easy to see that this "3" should go to position $\boldsymbol{p}(3)+\boldsymbol{c}(3) = 7$ of $\boldsymbol{k}_{\bar{B}_t}^{[1]}$. In other words, $\boldsymbol{k}_{\bar{B}_t}(1)$ is mapped to $\boldsymbol{k}_{\bar{B}_t}^{[1]}(7)$ by the permutation, hence $\boldsymbol{\sigma}^{[1]}(7) = 1$. The following *for* loop demonstrates how this works in general. Given this $\boldsymbol{\sigma}^{[1]}$, permutating $\boldsymbol{k}_{\bar{A}_s}$ for $\boldsymbol{k}_{\bar{A}_s}^{[1]}$ is trivial.

```
second scanning of k_{B̄_t}
for i = 1 : n
    read in the i^th element
    v = k_{B̄_t}(i)
    increment counter
    c(v) += 1
    write i to element (p(v) + c(v)) of σ^[1]
    σ^[1](p(v) + c(v)) = i
```

Binary MSD radix sort for secondary sort

In secondary sort, $\boldsymbol{k}_{\bar{A}_s}^{[1]}$ is a length-n integer vector taking values $1,\ 2,\ \cdots,\ m_{\bar{A}_s}$ and there is $n \gg m_{\bar{A}_s}$. The vector is then cut into $m_{\bar{B}_t}$ bins and bin-wise sort is applied on each bin to sort each segment in ascending order. Suppose the i^{th} segment of $\boldsymbol{k}_{\bar{A}_s}^{[1]}$ has length n_i and takes positive integer values no larger than m_i, then it is possible that $m_i > n_i$ or even $m_i \gg n_i$. As a result, counting sort with $O(n_i)$ time complexity and $O(m_i)$ space complexity is very inefficient. It is a better idea to resort to other sorting methods with $O(n_i \log(n_i))$ complexity on average, say quicksort and radix sort. Binary MSD radix sort is a simple variant of these algorithms. In the following I will demonstrate how this method works via a toy example.

Consider a length-15 integer vector below and its bit representation:

6,	2,	6,	3,	3,	3,	5,	1,	2,	3,	6,	6,	5,	3,	2
110,	010,	110,	011,	011,	011,	101,	001,	010,	011,	110,	110,	101,	011,	010

The maximum value in the vector is 6, so $\lceil \log_2(6+1) \rceil = 3$ bits are sufficient for bit representation and the most significant digit is 3. The sorting algorithm first examines the 3rd digit (counting from right to left) of those binaries, and sorts them by this digit. In the following, this digit is displayed in black, while the rest of the digits are displayed in gray. Note the way that sorting is done: the vector is scanned from two sides in opposing directions. From left to right, the first "1" encountered is displayed in red; from right to left, the first "0" encountered is displayed in blue. The corresponding two numbers are swapped if the red digit "1" is before (or to the left of) the blue digit "0". This process goes on until the two flagged digits cross each other somewhere in the middle. In this example, three swaps are made to reach this point.

110	010	110	011	011	011	101	001	010	011	110	110	101	011	010
010	010	110	011	011	011	101	001	010	011	110	110	101	011	110
010	010	011	011	011	011	101	001	010	011	110	110	101	110	110
010	010	011	011	011	011	011	001	010	101	110	110	101	110	110

Now the vector can be broken into two pieces at this crossover point: all numbers in the left piece have their 3rd digit being 0 and all numbers in the right piece have their 3rd digit being 1. The sorting algorithm then examines the 2nd digit on both pieces separately. In this example, it turns out that after one swap, a crossover point is hit in both pieces.

010	010	011	011	011	011	011	001	010	101	110	110	101	110	110
001	010	011	011	011	011	011	010	010	101	101	110	110	110	110

Now each piece can be further split into two, giving 4 pieces in total. The algorithm now examines the 1st digit on each piece separately. The first piece only has one single number hence is readily sorted; nothing needs be done further. The third and the fourth pieces are also readily sorted. Only the second piece needs further sorting.

001	010	011	011	011	011	011	010	010	101	101	110	110	110	110
001	010	010	011	011	011	011	010	011	101	101	110	110	110	110
001	010	010	010	011	011	011	011	011	101	101	110	110	110	110

The 1st digit is already the rightmost (least significant) digit, so the algorithm will terminate. The whole vector is now sorted, and I will now convert bit representation back to integers.

001,	010,	010,	010,	011,	011,	011,	011,	011,	101,	101,	110,	110,	110,	110
1,	2,	2,	2,	3,	3,	3,	3,	3,	5,	5,	6,	6,	6,	6

Here are a few remarks on this binary MSD radix sort.

- The algorithm is named with "binary" not just because it examines data in their binary format, but also because it scans the vector from two sides and eventually subdivides the vector into two. In this way, the algorithm can naturally be implemented recursively.

- The algorithm is an "in-place" algorithm. In the end of the algorithm, the initially unsorted vector will be overwritten by the sorted one.
- Permutation index vector can be generated alongside the sorted vector. This only requires that (in this example of 15 data) an index vector 1, 2, ..., 15 be initialized, and its elements swapped in the same way as elements of the vector are swapped. In fact, this can be more general. For example, in secondary sort, $\boldsymbol{\sigma}^{[1]}$ can be used instead of 1, 2, ..., n, and $\boldsymbol{\sigma}^{[2]}$ will overwrites $\boldsymbol{\sigma}^{[1]}$ when the sorting completes.

9.3.3 Dealing with sparse $\bar{\boldsymbol{W}}$

Now I am going to address another key issue in `bamboos`: dealing with the possible sparsity in $\bar{\boldsymbol{W}}$. Three questions need be answered:

1. How to construct / represent / store $\bar{\boldsymbol{W}}$ as a sparse matrix?
2. How to do matrix-matrix multiplication that involves a sparse $\bar{\boldsymbol{W}}$?
3. When is using a sparse $\bar{\boldsymbol{W}}$ advantageous to using a dense $\bar{\boldsymbol{W}}$?

Answering the first two questions just requires some basic knowledge of sparse matrices. In particular, neither question is performance related, as for example, there is no problem in representing / storing a realistic dense matrix using sparse matrix representation / storage method. The matter is whether doing so is worthwhile, which is exactly the aim of the last question.

Compressed column storage

There are undoubtedly many storage format for a sparse matrix. For a general sparse matrix without any special sparse structure[1], common storage format include triplet storage format, compressed row storage (CRS) format and compressed column storage (CCS) format. There is no "right" storage format, but only the convenient format that fits into the application context. For example, the run-length encoding and bin aggregation following the nested sorting readily yileds the triplet storage format. Consider the example in Figure 9.6, this is

- $\boldsymbol{i} = (2,\ 3,\ 1,\ 2,\ 1,\ 3,\ 4,\ 1,\ 4)$;
- $\boldsymbol{j} = (1,\ 1,\ 2,\ 2,\ 3,\ 3,\ 3,\ 4,\ 4)$;
- $\boldsymbol{x} = (x_1,\ x_2,\ x_3,\ x_4,\ x_5,\ x_6,\ x_7,\ x_8,\ x_9)$,

implying that $\bar{\boldsymbol{W}}(\boldsymbol{i}(v),\ \boldsymbol{j}(v)) = \boldsymbol{x}(v)$, $v = 1,\ 2,\ \cdots,\ 9$ where $\boldsymbol{x}(v)$ is the result of bin aggregation. It is also straightforward to obtain the CCS format by replacing $\boldsymbol{j}$ with the cumulative number of non-zero elements up to columns 0, 1, 2, 3, 4 (in other words, the lag-1 difference of $\boldsymbol{j}$ gives the number of non-zero elements for each column):

- $\boldsymbol{i} = (2,\ 3,\ 1,\ 2,\ 1,\ 3,\ 4,\ 1,\ 4)$;

[1] "Special" means that there is a certain pattern for zero and / or non-zero elements. Sparse matrices of special sparse structure include for example, block diagonal matrices and banded matrices. In fact, this thesis has already seen these sparse matrices. The $\boldsymbol{S}_{\boldsymbol{\lambda}}$ and its additive components $\tilde{\boldsymbol{S}}_{\boldsymbol{l}}$ in REML estimation (see equation (4.3) in §4.3) are block diagonals, and the diagonal or tri-diagonal weight matrix $\boldsymbol{W}$ are banded matrices.

- $\boldsymbol{j} = (0,\ 2,\ 4,\ 7,\ 9)$;
- $\boldsymbol{x} = (x_1,\ x_2,\ x_3,\ x_4,\ x_5,\ x_6,\ x_7,\ x_8,\ x_9)$.

$\boldsymbol{j}$ is often called a vector of *column pointers*, because the first non-zero element (and its row index) in column v can be located by skipping the first $\boldsymbol{j}(v)$ values in $\boldsymbol{x}$ (and $\boldsymbol{i}$).

Essentially the triplet storage format still interprets the example $\bar{\boldsymbol{W}}$ as a 4×4 matrix, although only making reference to its non-zero elements. This makes a sparse matrix human-readable, but is not convenient for practical operations with the sparse matrix like matrix-vector and matrix-matrix multiplications. These operations access the sparse matrix by row or column, thus require an easy way to extract non-zero values in $\boldsymbol{x}$ into row or column vectors. The CCR and CCS format are just for this purpose. For example, it is straightforward to extract column vectors (as compressed sparse vectors) from CCS format of the example $\bar{\boldsymbol{W}}$:

- skipping $\boldsymbol{j}(1) = 0$ elements in $\boldsymbol{i}$ and $\boldsymbol{x}$ and reading in $\boldsymbol{j}(2) - \boldsymbol{j}(1) = 2$ elements extracts column 1 as a compressed sparse vector: $\boldsymbol{i}^{[1]} = (2,\ 3)$, $\boldsymbol{x}^{[1]} = (x_1,\ x_2)$;
- skipping $\boldsymbol{j}(2) = 2$ elements in $\boldsymbol{i}$ and $\boldsymbol{x}$ and reading in $\boldsymbol{j}(3) - \boldsymbol{j}(2) = 2$ elements extracts column 2 as a compressed sparse vector: $\boldsymbol{i}^{[2]} = (1,\ 2)$, $\boldsymbol{x}^{[2]} = (x_3,\ x_4)$;
- skipping $\boldsymbol{j}(3) = 4$ elements in $\boldsymbol{i}$ and $\boldsymbol{x}$ and reading in $\boldsymbol{j}(4) - \boldsymbol{j}(3) = 3$ elements extracts column 3 as a compressed sparse vector: $\boldsymbol{i}^{[3]} = (1,\ 3,\ 4)$, $\boldsymbol{x}^{[3]} = (x_5,\ x_6,\ x_7)$;
- skipping $\boldsymbol{j}(4) = 7$ elements in $\boldsymbol{i}$ and $\boldsymbol{x}$ and reading in $\boldsymbol{j}(5) - \boldsymbol{j}(4) = 2$ elements extracts column 4 as a compressed sparse vector: $\boldsymbol{i}^{[4]} = (1,\ 4)$, $\boldsymbol{x}^{[4]} = (x_8,\ x_9)$.

In `bamboos`, the primary operation with a sparse matrix is matrix-matrix multiplication like $\bar{\boldsymbol{A}}'\bar{\boldsymbol{W}}$ and $\bar{\boldsymbol{W}}\bar{\boldsymbol{B}}$. Using CCS format for $\bar{\boldsymbol{W}}$ is thus more helpful. In the next section, I will sketch algorithms for both matrix multiplications.

Matrix multiplication with sparse $\bar{\boldsymbol{W}}$

Matrix-matrix multiplications $\tilde{\boldsymbol{A}} = \bar{\boldsymbol{A}}'\bar{\boldsymbol{W}}$ and $\tilde{\boldsymbol{B}} = \bar{\boldsymbol{W}}\bar{\boldsymbol{B}}$ in `bamboos` are not typical sparse matrix multiplications, because of the following.

- Only $\bar{\boldsymbol{W}}$ is sparse, while $\bar{\boldsymbol{A}}$ or $\bar{\boldsymbol{B}}$ is generally dense. There are exceptions of course. The packed design matrix for an i.i.d. random effect is an identity matrix, which is sparse. However, in this situation there is no need to even perform matrix multiplication. In practice this special case can be recognized and dealt with separately.
- The resulting matrix $\tilde{\boldsymbol{A}}$ and $\tilde{\boldsymbol{B}}$ are generally dense. This comes from the fact that each row or column of $\bar{\boldsymbol{W}}$ has at least one non-zero elements (To see this, just note that the discretization index vectors $\boldsymbol{k}_{\bar{A}}$ and $\boldsymbol{k}_{\bar{B}}$ contain all positive integers from 1 to $m_{\bar{A}}$ and from 1 to $m_{\bar{B}}$). Then in $\tilde{\boldsymbol{B}} = \bar{\boldsymbol{W}}\bar{\boldsymbol{B}}$, each row of $\tilde{\boldsymbol{B}}$ is a linear combination of rows of $\boldsymbol{B}$. Due to the previous fact, for any row of $\tilde{\boldsymbol{B}}$, there will be at least one row of $\boldsymbol{B}$ added to that row. Thus if $\boldsymbol{B}$ is dense, $\tilde{\boldsymbol{B}}$ is dense. That $\tilde{\boldsymbol{A}}$ is dense follows similarly.

These two characteristics make both the matrix-matrix multiplications very easy to program (without using any standard sparse linear algebra libraries). For example, consider programming $\tilde{\boldsymbol{A}} = \bar{\boldsymbol{A}}'\bar{\boldsymbol{W}}$, or $\tilde{\boldsymbol{A}}(u,\ v) = \sum_k \bar{\boldsymbol{A}}(k,\ u)\bar{\boldsymbol{W}}(k,\ v)$, then

```
initialize the resulting matrix
Ã = zeros(p_Ā, m_B̄)
for v = 1 : m_B̄
    number of non-zero elements in W̄(, v)
    r = j(r + 1) − j(r)
    offset for reading i and x
    s = j(r)
    for u = 1 : p_Ā
        initialize an accumulator
        a = 0
        loop over non-zero entries of W̄(, v)
        for k = 1 : r
            a += Ā(i(s + k), u)x(s + k)
        write-back
        Ã(u, v) = a
```

```
initialize the resulting matrix
B̃ = zeros(m_Ā, p_B̄)
for v = 1 : p_B̄
    for k = 1 : m_B̄
        linear combination coefficient
        b = B̄(k, v)
        number of non-zero elements in W̄(, k)
        r = j(k + 1) − j(k)
        offset for reading i and x
        s = j(k)
        loop over non-zero entries of W̄(, k)
        for u = 1 : r
            B̃(i(s + u), v) += x(s + u)b
```

Figure 9.9: Matrix multiplication $\tilde{\boldsymbol{A}} = \bar{\boldsymbol{A}}'\bar{\boldsymbol{W}}$ (left panel) and $\tilde{\boldsymbol{B}} = \bar{\boldsymbol{W}}\bar{\boldsymbol{B}}$ (right panel) when $\bar{\boldsymbol{W}}$ is sparse and stored with CCS format: $\boldsymbol{i}$ is the vector of row index, $\boldsymbol{j}$ is the vector of column pointers and $\boldsymbol{x}$ is the vector of $\bar{\boldsymbol{W}}$'s non-zero entries.

- it is sufficient to only check if $\bar{\boldsymbol{W}}(k,\ v)$ is zero or not when computing the inner product, because for any non-zero $\bar{\boldsymbol{W}}(k,\ v)$, the corresponding $\bar{\boldsymbol{A}}(k,\ u)$ is always non-zero;
- there is no need to trace the computation to judge whether $\tilde{\boldsymbol{A}}$ is dense or sparse hence what storage format should be used for it, because it is known to be dense.

Figure 9.9 sketches the algorithms for both matrix multiplications. Note that $\tilde{\boldsymbol{B}} = \bar{\boldsymbol{W}}\bar{\boldsymbol{B}}$ or $\tilde{\boldsymbol{B}}(u,\ v) = \sum_k \bar{\boldsymbol{W}}(u,\ k)\bar{\boldsymbol{B}}(k,\ v)$ is not programmed with v-u-k (outermost to innermost) loop ordering, but v-k-u loop ordering. This is because the former arrangement requires CRS format not CCS format.

"Dense-sparse switch" in `bamboos`

The nested sorting, run-length encoding and bin aggregation yields CCS format for $\bar{\boldsymbol{W}}$, making it equally straightforward to construct $\bar{\boldsymbol{W}}$ as dense or sparse, hence it is possible to leave the construction option as a run-time decision. The critical quantity for the decision is $K_{st}^{[1]}$, the number of non-zero elements in $\bar{\boldsymbol{W}}$ (the superscript is a reminder that I have been discussing diagonal $\boldsymbol{W}$), which is known from the last element of the vector of column pointers. A threshold value between $[0, 1]$ can be set for $\mathcal{D}_{st}^{[1]} = K_{st}^{[1]}/m_{\bar{A}_s}m_{\bar{B}_t}$, the proportion of non-zero elements in $\bar{\boldsymbol{W}}$, which is a measure of the density of the matrix.

- If $\mathcal{D}_{st}^{[1]}$ is above this threshold, $\bar{\boldsymbol{W}}$ should be constructed as dense (see Figure 9.7). Subsequent dense matrix-matrix multiplications can use level-3 BLAS for high-performance computing.
- If $\mathcal{D}_{st}^{[1]}$ drops below this threshold, $\bar{\boldsymbol{W}}$ should be constructed as sparse. A length-$K_{st}^{[1]}$ vector is initialized, then bin aggregation will aggregate weights $\dot{\boldsymbol{w}}_{\boldsymbol{gh}}$ onto this vector. A sparse matrix multiplication is first computed using algorithms in Figure 9.9, then all subsequent matrix-matrix multiplications are dense and level-3 BLAS can be used for high-performance computing.

The threshold therefore acts like a switch between dense construction and sparse construction. Fixing the threshold at 0 and 1 respectively enforce dense construction and sparse construction.

The threshold is a performance-related factor. It is not difficult to understand the following cases.

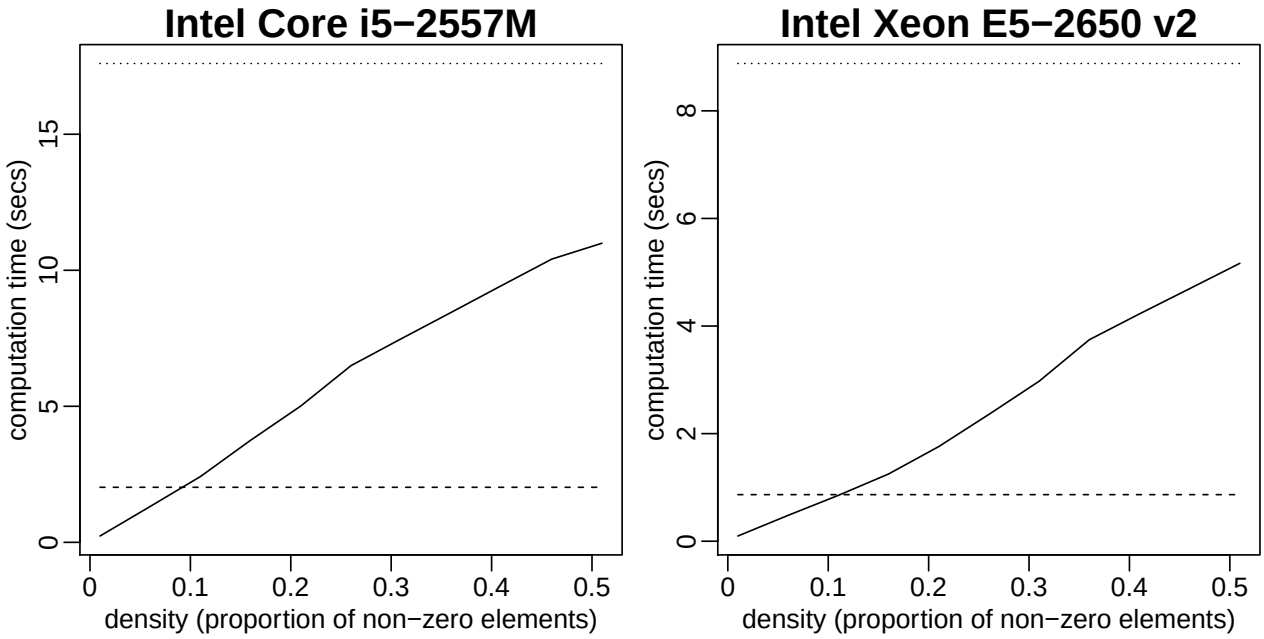


Figure 9.10: An empirical search for a reasonable threshold on Intel Core i5-2557M and Intel Xeon E5-2650 v2 (see Appendix A for hardware information). $\bar{\boldsymbol{W}}$ is constructed as a 2000 $\times$ 2000 random sparse matrix, with density $\mathcal{D}_{st}^{[1]}$ (x-axis) taking trial values 0.01, 0.06, 0.11, ..., 0.51. In both panels, the y-axis is the mean computation time of $\boldsymbol{W}\bar{\boldsymbol{B}}$ and $\bar{\boldsymbol{A}}'\boldsymbol{W}$, where $\bar{\boldsymbol{A}}$ and $\bar{\boldsymbol{B}}$ are 2000 $\times$ 2000 random dense matrices. Three computation methods are used: solid line for sparse method, i.e., algorithms in Figure 9.9; dashed line for dense method with OpenBLAS; dotted line for dense method with reference BLAS. On both machines, the solid line intersects the dashed line near 0.1, thus this is a reasonable threashold value for dense-sparse switch when an optimized BLAS is used.

- When $\mathcal{D}_{st}^{[1]}$ is big, i.e., when $\bar{\boldsymbol{W}}$ is realistically dense, doing matrix multiplication with the sparse method will induce great overhead due to index loopup, hence the dense method will outperform the sparse method.
- When $\mathcal{D}_{st}^{[1]}$ is small, i.e., when $\bar{\boldsymbol{W}}$ is realistically sparse, doing matrix multiplication with the dense method will waste substantial amount of time doing computation with zeros, hence the sparse method will outperform the dense method.

There must exist a crossover $\mathcal{D}_{st}^{[1]}$, and in theory this will be the optimal threshold. However, there is no other ways than practical benchmarking to make a proper guess on this value. In addition, this value is likely to depend on the following factors:

1. the machine where computation is performed;
2. the BLAS distribution that is used (it is not difficult to predict that with an optimized BLAS in place of the reference BLAS, it is more difficult for the sparse method to beat the dense method).

Figure 9.10 conducts an empirical search for a reasonable threshold on Intel Core i5-2557M and Intel Xeon E5-2650 v2 (see Appendix A for hardware information). It appears that 0.1 is a reasonable choice when an optimized BLAS (OpenBLAS in the experiment) is used. This threshold is currectly adopted in `bamboos`.

9.3.4 Tri-diagonal $\boldsymbol{W}$

Methods in the last section (§9.3) still works when the weight matrix $\boldsymbol{W}$ is tri-diagonal weight, by dealing its three diagonals separately. For example, the nested sorting will be applied to ($\boldsymbol{k}_{\bar{A}}$, $\boldsymbol{k}_{\bar{B}}$, $\boldsymbol{w}_{\boldsymbol{0}}$),

$(\boldsymbol{k}_{\bar{A}}(1:(n-1)),\ \boldsymbol{k}_{\bar{B}}(2:n),\ \boldsymbol{w}_1)$ and $(\boldsymbol{k}_{\bar{A}}(2:n),\ \boldsymbol{k}_{\bar{B}}(1:(n-1)),\ \boldsymbol{w}_{-1})$ independently. Bin aggregation is applied in each case, producing three contingency matrices. They are then added together for the final $\bar{\boldsymbol{W}}$.

9.4 Experimenting `bamboos` on daily logBS model

Previous sections have been a thorough demonstration of the design of `bamboos`. Now it is time to assess how worthwhile `bamboos` is in practical large GAM fitting. In this section, I will experiment `bamboos` on the test daily logBS model, model 7.1, and make performance comparison with the old discrete algorithm.

Recall from §8.6 that to compute $\boldsymbol{X'WX}$,

- 16-threaded "online" pseudo QR reduction with OpenBLAS takes 50 minutes;
- 16-threaded old discrete computation takes 34 minutes.

Application of `bamboos` shows that it takes 12 minutes, with just a single thread. That is, `bamboos` is so fast that even its serial computing performance has beaten previous methods with the aid of massively parallel computing. `bamboos` algorithms can be further parallelized, by parallelizing the computation within T × S, S × T and T × T blocks because of the following. Referring to Figure 9.4, different iterations are embarassingly parallel; in addition, they have the same computational complexity so load balancing is guaranteed. At it turns out, `bamboos` has very good parallel scaling. Using 4 threads, computation of $\boldsymbol{X'WX}$ completes in 3.3 minutes. And there is no need to use more threads.

A big advantage of `bamboos`, is that the more data there are, the faster it is compared with previous computation methods. This is what has been theoretically concluded from the $n \to \infty$ case analysis in §9.2.3. To verify this in practice, I will subsample (with replacement) the Monday logBS dataset for 1×10^6, 2×10^6, ..., 11×10^6 data and fit the test model 7.1. I will just compare `bamboos` (serial computing) with the old discrete method (16-threaded computing). The result is sketched in Figure 9.11. It can be seen that when data are 10 times as more, computation time for old discrete method is about 10 times as longer, but computation time for `bamboos` is just about less than 8 times as longer. As another test of `bamboos`, consider dropping the three-way interactions (three-margin tensor product splines) f_{16} and f_{17} from the test model, and redo the above experiment. This time it can be seen that when data are 15 times as more, computation time for `bamboos` is less than 7 times as longer.

9.5 Summary

In this Chapter I have developed a new computational engine, `bamboos`, for the matrix cross-product $\boldsymbol{X'WX}$, when the submatrices of $\boldsymbol{X}$ corresponding to individual GAM terms in a linear predictor $\boldsymbol{\eta} = \boldsymbol{X_0\beta_0} + \sum_{i=1}^{l} \boldsymbol{X_i\beta_i}$ are stored in a packed storage format from covariate discretization (described in §8.2). The method successfully boosts the previously successful discrete algorithms in Chapter 8 (and published in Wood et al. (2017)) by another magnitude.

`bamboos` is fast for the following reasons.

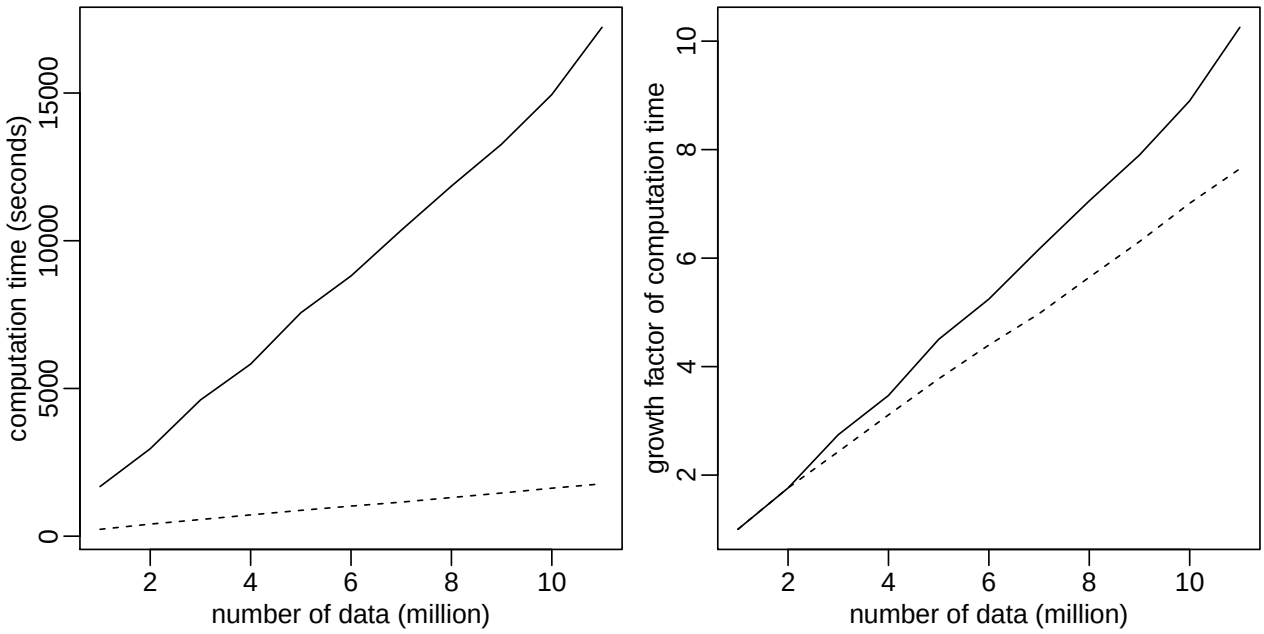


Figure 9.11: Benchmarking `bamboos` (serial computing) and old discrete method (16-threaded) on Intel E5-2650 v2 (see Appendix A for hardware information) by fitting test daily logBS model to 1×10^6, 2×10^6, ..., 11×10^6 data. The left panel shows the computation time for both methods (solid: old discrete; dashed: `bamboos`). For 1 million data, old discrete method takes 1680 seconds and `bamboos` takes 230 seconds; for 11 million data, old discrete method takes 17225 seconds and `bamboos` takes 1772 seconds. The right panel shows the growth factor of computation time for both methods (solid: old discrete; dashed: `bamboos`). It can be seen that when data are 10 times as more, computation time for old discrete method is about 10 times as longer, but computation time for `bamboos` is just about less than 8 times as longer.

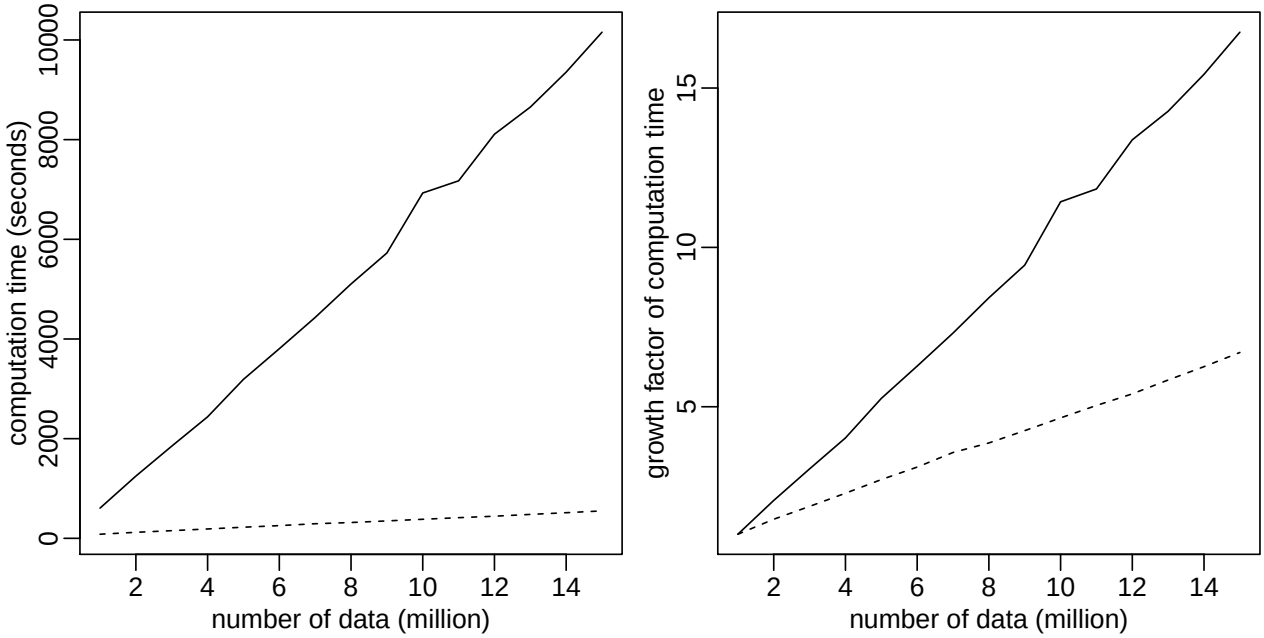


Figure 9.12: Benchmarking `bamboos` (serial computing) and old discrete method (16-threaded) on Intel E5-2650 v2 (see Appendix A for hardware information) by fitting test daily logBS model (dropping f_{16} and f_{17}) to 1×10^6, 2×10^6, ..., 15×10^6 data. The left panel shows the computation time for both methods (solid: old discrete; dashed: `bamboos`). For 1 million data, old discrete method takes 606 seconds and `bamboos` takes 82 seconds; for 15 million data, old discrete method takes 10155 seconds and `bamboos` takes 550 seconds. The right panel shows the growth factor of computation time for both methods (solid: old discrete; dashed: `bamboos`). It can be seen that when data are 15 times as more, computation time for old discrete method is about 15 times as longer, but computation time for `bamboos` is just about less than 7 times as longer.

1. Its fundamental design idea is superior to that of the old discrete methods. As is demonstrated in §9.1, `bamboos` algorithms are block-oriented, rich in matrix-matrix multiplications. By contrast, the old discrete algorithms are vector-oriented, rich in matrix-vector operations (like unpacking and aggregation). Therefore, `bamboos` has much better opportunity to benefit from optimized BLAS.

2. While the advantage of `bamboos` is not immediately clear from the upper bound of its FLOP count when compared with the old discrete algorithms (see §9.2), it gains substantial practical computational savings due to sparsity. Therefore, the speedup of `bamboos` is very often, huge, as is seen from the benchmarking in §9.4. Note that `bamboos` features a run-time decision between doing dense and sparse computations. This is very appealing, since users do not need to make judgement themselves. Other `R` packages like `glm4` that is successful in fitting large GLMs does require an exclusive user-specification on dense or sparse methods.

3. `bamboos` does not completely eliminate vector-oriented operations. The weights aggregation is a typical example, and in fact in the $n \to \infty$ limiting case, it will dominate `bamboos` computations. However, the nested sorting, run-length encoding and bin aggregation strategy makes such operation as fast as possible by optimizing memory access. The practical gains of these optimizations can be seen by comparing `bamboos` with its adapted version implemented in `mgcv` from version 1.8-25 (2018-10-26): `bamboos` is 6 times as fast in computing $\boldsymbol{X'WX}$ for the test daily logBS model, model 7.1.

Basically, `mgcv` has choosen a balance point between algorithmic efficiency and coding complexity. Admittedly, the nested sorting and sparse representation of $\bar{\boldsymbol{W}}$ add substantial difficulty in coding. That `bamboos` tries to accurately determine the degree of sparsity / density of $\bar{\boldsymbol{W}}$ (see quantities $K_{st}^{[1]}$ and $\mathcal{D}_{st}^{[1]}$), choose a near optimal dense-sparse switch, and optimize every memory access it sees is a heavy burden for its development. This is particulary true if $\boldsymbol{W}$ is tri-diagonal. `mgcv` uses a simpler heuristic on the sparsity of $\bar{\boldsymbol{W}}$ from the crude relationship between n and $m_{\bar{A}_s} m_{\bar{B}_t}$.

- If $m_{\bar{A}_s} m_{\bar{B}_t} < n$, the matrix is seen dense, and algorithm in the right panel of Figure 9.1a is used.
- If $m_{\bar{A}_s} m_{\bar{B}_t} > n$, the matrix is seen sparse. Yet `mgcv` opts to skip the explicit sparse construction of $\bar{\boldsymbol{W}}$, computing $\bar{\boldsymbol{A}}'\bar{\boldsymbol{W}}$ or $\bar{\boldsymbol{W}}\bar{\boldsymbol{B}}$ directly by matrix row aggregation.

In this way, the coding complexity is substantially reduced, but neither option above is optimal. For example, in the first case, the random memory access in weights aggregation is not be resolved, so the performance penalty observed in Figure 9.8 will have negative effect. In addition, $m_{\bar{A}_s} m_{\bar{B}_t} < n$ is a too optimistic bet on that $\bar{\boldsymbol{W}}$ is dense. For example, when $m_{\bar{A}_s} m_{\bar{B}_t} = 0.95n$, $\bar{\boldsymbol{W}}$ is realistically likely to be sparse hence using dense computation method will lose some efficiency. Nevertheless, the adapted implementation in `mgcv` preserves many good features of original `bamboos` design, thus is still significantly faster than the old discrete algorithms (hence very useful). A paper "Faster model matrix crossproducts for large generalized linear models with discretized covariates" has been submitted to *Statistics and Computing* based on this implementation.

It is worth pointing out that while `bamboos` is originally designed for fitting large GAMs, broadly speaking, it offers advantages for any regression model in which each covariate results in a term with several associate model matrix columns: models containing several factor variables are an obvious example beyond smooth regression models. However, there is still a long way to go if `bamboos` is to be developed into an `R` package (like `Matrix` package) that provides generic computation methods. At the moment, I am hoding an unpublished (not even on GitHub, etc) `bamboos` package for methods described in this Chapter.

Discussion

What is this thesis about? What is my PhD about? It is very difficult to summarize. Pessimistically speaking, it could be that I have attempted many things, but only half of them were made into this thesis, among which I have only been truely successful with one thing: `bamboos`.

`bamboos` offers an extremely efficient computation method for large GAM (or GLM) fitting. Strictly speaking, `bamboos` is only a far end of GAM fitting method for large datasets. It is built upon at least two critical previous research work.

- The idea of breaking up GAM fitting into model matrix reduction and smoothing parameter selection, and the application of "online" algorithm to the former.
- The discovery of the discrete characteristic in Black Smoke dataset, the proposal to thus store data in a compressed format, and the initial set of discrete algorithms that are able to do simple operations like vector unpacking and vector aggregation on data stored in such format.

The first idea had just been around when I started my PhD. The `bam` function implemented it. In today's view, it was a very crude implementation, with many computations arranged at `R`-level rather than `C`-level. But this is not surprising. Back at that stage, ensuring correctness of computations and validility of the estimation idea was the top priority. There were thus many opportunities to optimize it. It was only known that it would be successful to overcome the memory bottlenect imposed by large datasets and large model matrices, but how efficient the method was in large GAM fitting was unknown. In fact, many other aspects had not been discovered as well, like how Cholesky factorization may handle rank-deficiency, and how psuedo QR reduction performs compared with QR reduction.

The Black Smoke dataset served a perfect example for testing the effectiveness of this fitting method. It was soon discovered that it was even difficult to fit a model to annual logBS of 20000 - 30000 data and 3000 - 5000 parameters. In reality, a predictive model for annual Smoke concentration is of little practical use; it is hard to find correlation between annual mean polution level and epidemic. Therefore, there was also a need to model logBS at a finner temporal resolution, which inevitably ended up with GAM fitting for substantially larger datasets. Meanwhile, some slowish preliminary model also suggested that there there exsits complicated interactions between variables and a reasonable model would have to involve a high number of paramters.

Building a satisfying statistical model is one question; being able to estimate one is another, and in some sense, more important. Without being able to fit a model first, there is no way to assess the quality that model is and how it may be improved. Based on this acknowledgment, the focus of the research increasingly shifted from building Black Smoke models to developing fast large GAM fitting methods, and it went further and further on this route. Eventually, this thesis became very strong on the computational aspect, but very weak in statistical modelling aspect.

Chapters of this thesis have been organized to highlight the early use of optimized BLAS for initial performance improvment of GAM fitting. However, the true story is, an optimized BLAS had not

been experimented even after the discrete algorithms were proposed, implemented and published. I started formal investigation on optimized BLAS when I tried to improve the performance of pivoted Cholesky factorization. Although after 6 - 12 months' investigation I was not able to roll out an implementation that consistently outperforms existing computation methods, I have learned a lot about how an optimized BLAS is designed, how to write fast `C` code, and how to read assembly output by a compiler and even write some assembly code for speed. Only by this stage, I then felt more competent in analyzing the performance of `mgcv`. What has been presented in Chapter 5 is really the result of my revisit to `bam`. I later found that the discrete method in Chapter 8, while substantially faster than "online" algorithms implemented with reference BLAS, is not competitive with that implemented with OpenBLAS. This greatly raised my curiosity, and I started to investigate discrete algorithms in depth again.

Note that it was already in the 5-th year of my PhD when I revisited the discrete algorithms. So while I was supposed to focus on writing my thesis, telling about the story of how the discrete algorithm was found and how daily Black Smoke model was built, I was spending substantial amount of time in researching the discrete algorithm. Initially I applied caching to a few steps of these discrete algorithms, but the gains was not impressive. I understood why: the old algorithms was vector-oriented, hard to attain high performance. The deadline for submitting my thesis was getting close, then I have to stop further investigation.

Personally speaking, I did not find anything I could be proud of when writing my thesis. The original idea of discrete algorithm was not proposed by me; my Black Smoke models were also very weak. Therefore, I did not have confidence that I could obtain a degree. The thesis was written very short, primarily because I did not see anything worthwhile to write (at least in my own opinion). To make my thesis longer, I tried hard, adding in many immature materials like using knots placement to suppress spatial-temporal prediction bias, developing some toy models for the dropout process of stations. However, these are not what I really want to do.

A month before the submission deadline, I suddently got some idea on how to arrange discrete algorithms in block-oriented fashion. I was very excited. I was sure this is going to be a novel idea. The basic idea of these algorithms is not very complicated; however, the nested sorting, run-length encoding, bin-aggregation, handling sparsity turns out complicated. But the more complicated it is, the more worthwhile it is to do. Before submission of the thesis I was able to establish theorectical superioirty of these methods. The first full implementation was not ready until my viva was near. I decided to place the development of these fast algorithms as my prominent contributions over the past 7 years.

I have been too narrowly focused on computational issues, and the building statistical models have been overlooked to some extent. In the revision / correction stage of the thesis, more prudent model development for logBS were added. These models turned out much more complicated than those in the original thesis. I did learn plenty of things in this process, for example,

- I implemented the golden-section search so that GAMs can be estimated together with AR(1) autocorrelation.
- I also learned to thin data to alleviate or even eliminate the effect of temporal autocorrelation in building GAMs for exploratory analysis.
- I also learned plenty of visualization methods for model checking.

At the same time, I also realized that it is very difficult to adequately model spatial autocorrelation in the data. I concluded that splines is more useful to remove long-range spatial trend, but less competent in modelling short range variability.

These revised model development altered my opinion of building daily logBS model. It is even difficult

to build an adequate model for thinned data (weekly logBS from each day of week) without temporal correlation, there is hardly any sense to move on further to a full daily model. If I would ever revisit Black Smoke models again, I would highly recommend first getting more daily covariate variables. The exploratory analysis methods presented in Chapter 3 will still be valid.

I honestly admitted that I am incompetent in building statistical models. However, I have provided fast model fitting methods, offering opportunities for those competent to investigate the dataset. I would still devote my future work on computational issues. Compared with model building, I feel that these matters are more likely to be firmly answered. Life is too short; I want to do something that is 100% answerable.

Appendix A

Hardware information

Although the computational methods reviewed and to be developed in this thesis are not targeted on any specific machine, there is still a need to introduce machines that are used for computations, profiling and benchmarking. Three machines are used in this thesis, and all of them come with Linux OS.

- Intel Core i5-2557M Sandy Bridge laptop. It has two CPU cores, each of which can work at a sustainable frequency of 1.1GHz. 4GB RAM is available, with 32KB L1 data cache for each core. There are two memory channels in total, each supporting DDR3 DIMM at 1333MHz, giving a maximum FSB bandwidth of 10.664 GB/s per channel. OpenBLAS is available on this machine for matrix computations;
- Intel Xeon E5-2650 v2 Ivy Bridge workstation. It has 16 CPU cores, each of which can work at a sustainable frequency of 2.6GHz. 128GB RAM is available, with 32KB L1 data Cache for each core. There are four memory channels in total, each supporting DDR3 DIMM at 1866 MHz, giving a maximum FSB bandwidth of 14.928 GB/s per channel. This machine is also the only one with NUMA feature, as its 16 cores sit on two CPU sockets. This machine is a node of University of Bath's HPC cluster *Balena*, launched in 2015. Both Intel MKL and OpenBLAS are available on this machine for matrix computations;
- Intel Core M-5Y71 Broadwell laptop. It has two CPU cores, each of which can work at a sustainable frequency of 1.3GHz. 8GB RAM is available, with 32KB L1 data cache for each core. There are two memory channels in total, each supporting LPDDR3 DIMM at 1600 MHz, giving a maximum FSB bandwidth of 12.8 GB/s per channel. OpenBLAS is available on this machine for matrix computations.

"Sustainable frequency" is a frequency that a CPU can sustain at during long-time intensive computations without overheating, in both single-threading and multi-threading application. All these machines come with Intel's "turbo boost" technology, which enables a higher CPU frequency for a short burst of time in single-threading application. Core i5-2557M has a "turbo" range $1.7 \sim 2.7$GHz, Xeon E5-2650 v2 has a "turbo" range $2.6 \sim 3.4$GHz, Core M-5Y71 has a "turbo" range $1.3 \sim 2.9$ GHz. CPU frequency in "turbo" range is not sustainable, as the excessive amount of heat it generates would eventually cause dynamic CPU frequency scaling to reduce the frequency. "Turbo boost" must be disabled to rule out the impact of unstable CPU frequency in benchmarking. Note that on Core i5-2557M, simply disabling "turbo boost" is not enough. The maximum 1.7GHz frequency below "turbo" range can still cause CPU temperature to reach the critical 86°C within 1 minutes. It turns out that 1.1GHz is its sustainable frequency.

Originally all machines have the capability for scheduling hyper-threading, but it is later disabled. This technology allows two threads to be scheduled on each CPU core, so it appears to the operating

system as well as users that there are two logical CPU cores per physical CPU core. However, logical cores have CPU affinity of 1, sharing all physical resources like CPU registers and CPU cache. In computationally intensive tasks like dense matrix computations, hyper-threading would hammer practical parallel scalability, because logical cores would be busy competing for physical resources and having two of them is hardly any better than having just one. Hyper-threading is more promising in sparse matrix computations which is more memory intensive, as if one thread has a cache miss in reading data from RAM, the other thread may have a cache hit so the CPU core can avoid being idle.

Bibliography

Anderson, E., Bai, Z., Bischof, C., Blackford, L. S., Demmel, J., Dongarra, J. J., Du Croz, J., Hammarling, S., Greenbaum, A., McKenney, A., and Sorensen, D. (1999). *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, 3 edition.

Anderson, E., Bai, Z., Bischof, C., Demmel, J. W., Dongarra, J. J., Croz, J. D., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. C. (1990). LAPACK: A portable linear algebra library for high-performance computers. Technical Report 20, LAPACK Working Note.

Anderson, E. and Dongarra, J. J. (1990). Implementation guide for LAPACK. Technical Report 18, LAPACK Working Note.

Augustin, N. H., Musio, M., von Wilpert, K., Kublin, E., Wood, S. N., and Schumacher, M. (2009). Modeling spatiotemporal forest health monitoring data. *Journal of the American Statistical Association*, 104(487):899–911.

Belitz, C. and Lang, S. (2008). Simultaneous selection of variables and smoothing parameters in structured additive regression models. *Computational Statistics and Data Analysis*, 53(1):61–81.

Bischof, C. and Loan, C. V. (1987). The wy representation for products of householder matrices. *SIAM Journal on Scientific and Statistical Computing*, 8(1):s2–s13.

Bowman, A. and Azzalini, A. (1997). *Applied Smoothing Techniques for Data Analysis: The Kernel Approach with S-Plus Illustrations*. Oxford Statistical Science Series. OUP Oxford.

Breslow, N. E. and Clayton, D. G. (1993). Approximate inference in generalized linear mixed models. *Journal of the American Statistical Association*, 88(421):9–25.

Burylov, I., Chuvelev, M., et al. (2007). Intel performance libraries: multi-core-ready software for numeric-intensive computation. *Intel technology journal*, 11(4):299–308.

Ciocco, A. and Thompson, D. (1961). A follow-up of donora ten years after: methodology and findings. *Am J Public Health Nations Health*, 51:155–164.

Claeskens, G., Krivobokova, T., and Opsomer, J. D. (2009). Asymptotic properties of penalized spline estimators. *Biometrika*, 96(3):529–544.

Cleveland, W. S. (1979). Robust locally weighted regression and smoothing scatterplots. *Journal of the American Statistical Association*, 74(368):829–836.

Cleveland, W. S. and Devlin, S. J. (1988). Locally weighted regression: An approach to regression analysis by local fitting. *Journal of the American Statistical Association*, 83(403):596–610.

Clifton, M. (1964). Air Pollution. *Proceedings of the Royal Society of Medicine*, 57(7):615–618.

Crainiceanu, C., Ruppert, D., and Wand, M. (2005). Bayesian analysis for penalized spline regression using winbugs. *Journal of Statistical Software, Articles*, 14(14):1–24.

Craven, P. and Wahba, G. (1979). Smoothing noisy data with spline functions. *Numerische Mathematik*, 31:377–403.

Croz, J. D. and Higham, N. J. (1992). Stability of methods for matrix inversion. *IMA Journal of Numerical Analysis*, 12(1):1–19.

Dadvand, P., Rushton, S., et al. (2011). Using spatio-temporal modelling to predict long-term exposure to Black Smoke at fine spatial and temporal scale. *Atomosphere Environment*, 45(3):659–664.

de Boor, C. (1978). *A Practical Guide to Splines*. Applied Mathematical Sciences. Springer-Verlag New York.

Demmel, J. W., Dongarra, J. J., Croz, J. D., Greenbaum, A., Hammarling, S., and Sorensen, D. C. (1987). Prospectus for the development of a linear algebra library for high-performance computers. Technical Report 1, LAPACK Working Note.

Demmel, J. W., Hoemmen, M., Hida, Y., and Riedy, E. J. (2009). Nonnegative diagonals and high performance on low-profile matrices from householder qr. *SIAM Journal on Scientific Computing*, 31(4):2832–2841.

Dongarra, J. J., Croz, J. D., Hammarling, S., and Duff, I. (1990). A set of level 3 basic linear algebra subprograms. *ACM transactions on mathematical software*, 16(1):1–17.

Dongarra, J. J., Croz, J. D., Hammarling, S., and Hanson, R. J. (1988). An extended set of basic of fortran basic linear algebra subprograms. *ACM transactions on mathematical software*, 14(1):1–17.

Drepper, U. (2007). What every programmer should know about memory. Technical report, Red Hat, Inc. `https://people.freebsd.org/~lstewart/articles/cpumemory.pdf`.

Duchon, J. (1977). *Constructive theory of functions of several variables*, volume 571, chapter Splines minimizing rotation-invariant semi-norms in Solobev spaces, pages 85–100. Springer Berlin Heidelberg.

Duersch, J. A. and Gu, M. (2017). Randomized qr with column pivoting. *SIAM Journal on Scientific Computing*, 39(4):C263–C291.

Eilers, P. H. C. and Marx, B. D. (1996). Flexible smoothing with B-splines and penalties. *Statistical Science*, 11(2):89–121.

Eilers, P. H. C. and Marx, B. D. (2003). Multivariate calibration with temperature interaction using two-dimensional penalized signal regression. *Chemometrics and intelligent laboratory systems*, 66(2):159–174.

Enea, M. (2009). Fitting linear models and generalized linear models with large data sets in R. In *book of short papers, conference on "Statistical Methods for the analysis of large data-sets"*, pages 411–414. Italian Statistical Society, Chieti-Pescara,.

Epperson, J. F. (1987). On the runge example. *American Mathematical Monthly*, 94(4):329–341.

Fahrmeir, L. and Lang, S. (2001). Bayesian inference for generalized additive mixed models based on markov random field priors. *Applied Statistics*, 50:201–220.

Fan, J. and Gijbels, I. (1996). *Local Polynomial Modelling and Its Applications*, volume 66 of *Monographs on Statistics and Applied Probability*. Chapman and Hall/CRC.

Fanshawe, T. R., Diggle, P. J., et al. (2008). Modelling spatio-temporal variation in exposure to particulate matter: a two-stage approach. *Environmetrics*, 19(6):549–566.

Faraway, J. J. (2004). *Linear models with R*. Texts in Statistical Science. Chapman and Hall/CRC.

Firket, J. (1936). Fog along the Meuse valley. *Transactions of the Faraday Society*, 32:1191–1194.

Golub, G. H. and Loan, C. F. V. (2013). *Matrix Computations*. The Johns Hopkins University Press, Baltimore, 4 edition.

Goto, K. and Geijn, R. V. D. (2008a). Anatomy of high-performance matrix multiplication. *ACM transactions on mathematical software*, 34(3).

Goto, K. and Geijn, R. V. D. (2008b). High-performance implementation of the level-3 BLAS. *ACM transactions on mathematical software*, 35(1).

Green, P. and Silverman, B. (1994). *Nonparametric Regression and Generalized Linear Models*, volume 58 of *Monographs on Statistics and Applied Probability*. Chapman and Hall.

Gu, C. (1992). Cross-validating non-gaussian data. *Journal of Computational and Graphical Statistics*, 1(2):169–179.

Gu, C. and Kim, Y. J. (2002). Penalized likelihood regression: general approximation and efficient approximation. *Canadian Journal of Statistics*, 34(4):619–628.

Gulliver, J., Morris, C., et al. (2011). Land use regression modeling to estimate historic (1962-1991) concentrations of Black Smoke and Sulphur Dioxide for Great Britain. *Environmental Science and Technology*, 45:3526–3532.

Gunnels, J. A., Henry, G. M., and Geijn, R. A. (2001). *ICCS 2001: International Conference San Francisco, CA, USA, May 28–30, 2001 Proceedings, Part I*, chapter A Family of High-Performance Matrix Multiplication Algorithms, pages 51–60. Springer Berlin Heidelberg.

Hall, P. and Opsomer, J. D. (2005). Theory for penalised spline regression. *Biometrika*, 92(1):105–118.

Handcock, M. S., Meier, K., and Nychka, D. (1994). Comment. *Journal of the American Statistical Association*, 89(426):401–403.

Harville, D. A. (1997). *Matrix algebra from a statistician's perspective*. Springer.

Hastie, T. and Tibshirani, R. (1986). Generalized additive models (with discussion). *Statistical Science*, 1:297–318.

Hastie, T. and Tibshirani, R. (1990). *Generalized Additive Models*. Chapman and Hall.

Hastie, T., Tibshirani, R., and Friedman, J. (2009). *Kernel Smoothing Methods*, pages 191–218. Springer New York, New York, NY.

Helwig, N. E. and Ma, P. (2016). Smoothing spline ANOVA for super large samples: scalable computation via rounding parameters. *preprint arXiv:1602.05208*.

Hennessy, J. L. and Patterson, D. (2011). *Computer Architecture: A Quantitative Approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier, 5 edition.

Kågström, B., Ling, P., and Loan, C. V. (1998). Gemm-based level 3 BLAS: High performance model implementations and performance evaluation benchmark. *ACM transactions on mathematical software*, 24(3):268–302.

Kammann, E. E. and Wand, M. P. (2003). Geoadditive models. *Applied Statistics*, 52(1):1–18.

Kauermann, G., Krivobokova, T., and Fahrmeir, L. (2009). Some asymptotic results on generalized penalized spline smoothing. *Journal of the Royal Statistical Society: Series B*, 71(2):487–503.

Kimeldorf, G. S. and Wahba, G. (1970). A correspondence between bayesian estimation on stochastic processes and smoothing by splines. *The Annals of Mathematical Statistics*, 41(2):495–502.

Kowarschik, M. and Weiß, C. (2003). *An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms*, pages 213–232. Springer Berlin Heidelberg, Berlin, Heidelberg.

Kress, R. (1996). *Numerical Analysis*, volume 181 of *Graduate Texts in Mathematics*. Springer-Verlag New York.

Krivobokova, T. and Kauermann, G. (2007). A note on penalized spline smoothing with correlated errors. *Journal of the American Statistical Association*, 102(480):1328–1337.

Lam, M. D., Rothberg, E. E., and Wolf, M. E. (1991). The cache performance and optimizations of blocked algorithms. *SIGPLAN Notices*, 26(4):63–74.

Lang, S., Umlauf, N., Wechselberger, P., et al. (2014). Multilevel structured additive regression. *Statistics and Computing*, 24(2):223–238.

Lawson, C. L., Hanson, R. J., Kincaid, D. R., and Krogh, F. T. (1979). Basic linear algebra subprograms for fortran usage. *ACM transactions on mathematical software*, 5(3):308–323.

Lee., D.-J. and Durbán, M. (2011). P-spline anova-type interaction models for spatio-temporal smoothing. *Statistical Modelling*, 11(1):49–69.

Li, Y. and Ruppert, D. (2008). On the asymptotics of penalized splines. *Biometrika*, 95(2):415–436.

Lindgren, F. and Rue, H. (2015). Bayesian spatial modelling with r-inla. *Journal of Statistical Software*, 63(19):1–25.

Lindgren, F., Rue, H., and Lindström, J. (2011). An explicit link between gaussian fields and gaussian markov random fields: the stochastic partial differential equation approach. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 73(4):423–498.

Loader, A. (2002). *Instruction manual: UK Smoke and Sulphur Dioxide Network*. Netcen, AEA Technology, Culham Science Centre.

Lucas, C. (2004). LAPACK-style codes for level 2 and 3 pivoted cholesky factorizations. Technical Report 161, LAPACK Working Note.

Martinsson, P.-G., Ort, G. Q., Heavner, N., and van de Geijn, R. (2017). Householder qr factorization with randomization for column pivoting (hqrrp). *SIAM Journal on Scientific Computing*, 39(2):C96–C115.

Marx, B. D. and Eilers, P. H. C. (1998). Direct generalized additive modeling with penalized likelihood. *Computational Statistics and Data Analysis*, 28:193–209.

Miller, A. J. (1992). Algorithm AS 274: Least Squares Routines to Supplement Those of Gentleman. *Journal of the Royal Statistical Society: Series C*, 41(2):458–478.

Ministry of Health (1954). *Mortality and morbidity during the London fog of December 1952*. HMSO, London.

Nadaraya, E. A. (1964). On estimating regression. *Theory of Probability & Its Applications*, 9(1):141–142.

Nocedal, J. and Wright, S. (2006). *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer.

Opsomer, J., Wang, Y., and Yang, Y. (2001). Nonparametric regression with correlated errors. *Statistical Science*, 16(2):134–153.

Parker, P. and Rice, J. (1985). Discussion of silverman (1985). *Journal of the Royal Statistical Society: series B*, 47(1):40–42.

Patterson, H. D. and Thompson, R. (1971). Recovery of inter-block information when block sizes are unequal. *Biometrika*, 58(3):545–554.

Perry, M. and Hollis, D. (2006). *The generation of monthly gridded datasets for a range of climatic variables over the United Kingdom*. Met Office, FitzRoy Road, Exeter, Devon, EX1 3PB, United Kingdom.

Perry, M., Hollis, D., and Elms, M. (2009). *The Generation of Daily Gridded Datasets of Temperature and Rainfall for the UK*. National Climate Information Centre, Met Office, FitzRoy Road, Exeter, Devon, EX1 3PB, United Kingdom.

Plummer, M. (2003). JAGS: A program for analysis of Bayesian graphical models using Gibbs sampling. In *Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003)*.

Quintana-Ortí, G., Sun, X., and Bischof, C. H. (1998). A BLAS-3 version of the QR factorization with column pivoting. *SIAM Journal on Scientific Computing*, 19(5):1486–1494.

Reinsch, C. H. (1967). Smoothing by spline functions. *Numerische Mathematik*, 10:177–183.

Reinsch, C. H. (1971). Smoothing by spline functions. ii. *Numerische Mathematik*, 16(5):451–454.

Reiss, P. T. and Ogden, T. R. (2009). Smoothing parameter selection for a class of semiparametric linear models. *Journal of the Royal Statistical Society: Series B*, 71(2):505–523.

Rue, H. and Held, L. (2005). *Gaussian Markov Random Fields: Theory and Applications*, volume 104 of *Monographs on Statistics and Applied Probability*. Chapman and Hall.

Ruppert, D., Wand, M. P., and Carroll, R. J. (2003). *Semiparametric Regression*. Cambridge University Press.

Shaddick, G. and Zidek, J. (2014). A case study in preferential sampling: Long term monitoring of air pollution in the UK. *Spatial Statistics*, 9.

Silverman, B. (1985). Some aspects of the spline smoothing approach to non-parametric regression curve fitting. *Journal of the Royal Statistical Society: series B*, 47(1):1–53.

Smith, M., Wong, C.-M., and Kohn, R. (1998). Additive nonparametric regression with autocorrelated errors. *Journal of the Royal Statistical Society. Series B (Statistical Methodology)*, 60(2):311–331.

Spiegelhalter, D. J., Thomas, A., Best, N. G., and Gilks, W. R. (1996). *BUGS: Bayesian inference Using Gibbs Sampling, Version 0.5, (version ii)*. MRC Biostatistics Unit, Cambridge.

Sven, H. and Craig, L. (2008). Updating the QR factorization and the least squares problem. Technical report, Manchester Institute for Mathematical Sciences, School of Mathematics, The University of Manchester. `http://eprints.ma.man.ac.uk/1192/1/qrupdating_12nov08.pdf`.

Terrain-50 (2015). *OS Terrain 50: user guide and technical specification*. Ordnance Survey, Adanac Drive, Southampton, SO16 0AS.

Tuerlinckx, F., Rijmen, F., Verbeke, G., and Boeck, P. (2010). Statistical inference in generalized linear mixed models: A review. *British Journal of Mathematical and Statistical Psychology*, 59(2):225–255.

van de Geijn, R. and Quintana-Ortí, E. (2008). *The Science of programming matrix computations*. Lulu Press, Inc.

Van Zee, F. G., Smith, T., Igual, F. D., Smelyanskiy, M., Zhang, X., Kistler, M., Austel, V., Gunnels, J., Low, T. M., Marker, B., Killough, L., and van de Geijn, R. A. (2016). The BLIS framework: Experiments in portability. *ACM Transactions on Mathematical Software*, 42(2):12:1–12:19.

Van Zee, F. G. and van de Geijn, R. A. (2015). BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Transactions on Mathematical Software*, 41(3):14:1–14:33.

Wahba, G. (1980). *Approximation Theory III*, chapter Spline bases, regularization, and generalized cross validation for solving approximation problems with large quantities of noisy data, pages 905–912. Academic Press.

Wahba, G. (1981). Spline interpolation and smoothing on the sphere. *SIAM Journal on Scientific and Statistical Computing*, 2(1):5–16.

Wahba, G. (1983). Bayesian confidence intervals for the cross validated smoothing spline. *Journal of the Royal Statistical Society: series B*, 45:133–150.

Wahba, G. (1990). *Spline Models for Observational Data*. Society for Industrial and Applied Mathematics.

Wand, M. P. and Jones, M. C. (1994). *Kernel smoothing*, volume 60 of *Monographs on Statistics and Applied Probability*. Chapman and Hall/CRC.

Wang, X., Shen, J., and Ruppert, D. (2011). On the asymptotics of penalized spline smoothing. *Electronic Journal of Statistics*, 5:1–17.

Wang, Y. (1998). Smoothing spline models with correlated random errors. *Journal of the American Statistical Association*, 93(441):341–348.

Watson, G. S. (1964). Smooth regression analysis. *Sankhyā: The Indian Journal of Statistics, Series A (1961-2002)*, 26(4):359–372.

Welford, B. P. (1962). Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420.

Whaley, R. C., Petitet, A. P., and Dongarra, J. J. (2000). Automated empirical optimization of software and the ATLAS project. Technical Report 147, LAPACK Working Notes.

Wood, S. (2016). Just another gibbs additive modeler: Interfacing jags and mgcv. *Journal of Statistical Software, Articles*, 75(7):1–15.

Wood, S., Li, Z., Shaddick, G., and Augustin, N. (2017). Generalized additive models for gigadata: modelling the UK Black Smoke Network daily data. *Journal of the American Statistical Association*, 112(519):1199–1210.

Wood, S. N. (2003). Thin plate regression splines. *Journal of the Royal Statistical society: Series B*, 65(1):95–114.

Wood, S. N. (2004). Stable and efficient multiple smoothing parameter estimation for generalized additive models. *Journal of the American Statistical Association*, 99(467):637–686.

Wood, S. N. (2006). Low-rank scale-invariant tensor product smooths for generalized additive mixed models. *Biometrics*, 62(4):1025–1036.

Wood, S. N. (2008). Fast stable direct fitting and smoothness selection for generalized additive models. *Journal of the Royal Statistical society: Series B*, 70(3):495–518.

Wood, S. N. (2011). Fast stable restricted maximum likelihood and marginal likelihood estimation of semiparametric generalized linear models. *Journal of the Royal Statistical society: Series B*, 73(1):3–36.

Wood, S. N. (2017). P-splines with derivative based penalties and tensor product smoothing of unevenly distributed data. *Statistics and Computing*, 27(4):985–989.

Wood, S. N., Bravington, M. V., and L., H. S. (2008). Soap film smoothing. *Journal of the Royal Statistical society: Series B*, 70(5):931–955.

Wood, S. N., Goude, Y., and Shaw, S. (2015). Generalized additive models for large data sets. *Journal of the Royal Statistical society: Series C*, 64(1):139–155.

Wood, S. N., Scheipl, F., and Faraway, J. (2013). Straightforward intermediate rank tensor product smoothing in mixed models. *Statistics and Computing*, 23(3):341–360.

Zhang, X., Wang, Q., and Zhang, Y. (2011). OpenBLAS: A high performance BLAS library on Loongson 3A CPU. *Journal of Software*, 22:208–216.

Zuur, A., Ieno, E., and Saveliev, A. (2014). *A Beginner's Guide to Generalised Additive Mixed Models with R*. Highland Statistics Limited.